Universitetet i Oslo
Institutt for informatikk

QGEN: A Python to Qt/C++ translator

Martin Stokland Jensen

Master Thesis

29th November 2004

Preface

This document is the documentation of my Master Thesis at the University of Oslo, Department of Informatics. It is mainly aimed at the censor who will, together with my counselors, ultimately grade my work. In addition to satisfying the censor, selected parts of the document will be of interest to programmers who want to dig a bit deeper than what the comments in the code will allow.

Some programming experience is expected if you are reading this. A few sections can get a bit technical and you are expected to be able to read a sentence like "Python uses dynamically bound, strongly typed variables" without having to consult a technical dictionary.

I will start by explaining the background for the work and why I am doing this. I will discuss the different technologies and languages used and explain what the program does and how it is used. The programming challenges are also addressed as well as an explanation of the techniques used in the code. Lastly, the work is summed up by telling what I have achieved and who this project should be of interest to.

I am writing this document in English because I would like to continue this work after the master thesis is done. In real world programming, I would make all code available on a web site so that other people who take an interest in the project could read the code and documentation and make suggestions on their own. They would be able to rewriting some bad code even or adding features and fixing bugs. For this to be a possibility all code and documentation should be in English.

I would like to thank to Ola Skavhaug and Hans Petter Langtangen for inspiration and guidance while working on this project. Their help has been invaluable, especially for motivation in the latter stage of the project when I more than once sat around the clock coding, writing and re-writing code segments and sections of this text.

Conventions

The code

When coding, I try to follow the guidelines laid out in the Python Enhancement Proposals (PEPs) from the Python website [11]. This is a collection of guidelines, rules, and tips for writing good code. It covers both C-code and Python-code and gives proposals on commenting and documenting as well as coding-tips. I will however - in accordance with the guidelines for the guidelines - break the rules occasionally in places where I feel it is appropriate. This is to ensure that the code is as readable as I feel I can make it.

As mentioned, the PEPs also cover commenting the code. I will comment all Python code so that I will be able to extract a complete PyDoc. This is included in appendix D. Explanatory comments within the code are usually written like this:

```
x = 10
#
# Comment explaining the reason for incrementing x by 20
#
x = x + 20
x = 8
```

This way I feel that the comments stand out a bit, creating welcoming space in the code and making it more readable.

I will also extract a complete documentation from the C++ header files. This can be found in appendix E

This document

All programming code will be written in `monospaced` font:

```
10 PRINT "HOME"
20 PRINT "SWEET"
30 GOTO 10
```

I am aware that all books, articles, and other writings concerning the programming language Python, are supposed to contain as many obscure references to the comedy of Monty Python [10] as possible. Even though this seems to be a well established convention in the programming community, I have not gone to great lengths to achieve this. A reference or two may show up, but I have to intended for this to be convention throughout the document.

Contents

Chapter 1

Introduction

The goal of this project is to create a program which automatically translates - or compiles - a program written in the language Python to Qt/C++. Qt/C++ is basically C++ using the Qt graphics library created by Trolltech [12]. The target code should be able to compile on a Sharp Zaurus SL-5500 PDA.

This project is a piece of a bigger project whose aim is to be able to compile Python programs on many different handheld units like PDAs (Personal Digital Assistants). The Zaurus, which is "my" PDA, runs Qtopia and Qt/C++ programs. The advantage with the Qt/C++ language is that it is able to run on a large number of platforms, and not limited to just the Zaurus. The code my program produces will therefore be virtually platform independent.

When code is generated for the PDA, graphics will have to be manipulated to fit on a much smaller screen. When using the Qt graphics library for an embedded device, special Qt classes need to be used instead of the standard Qt/X11 classes.

1.1 Starting the project

When I first started this project I had had little experience with Python. My first thought was to write a few programs and see what could be done to analyze the code. There were three ways, as I saw it, to solve this task.

The first was using regular expressions to read and analyze Python source code and output it as Qt/C++ code. This seemed to be the *hard way* of doing this. All lines of code would have to examined and taken apart by regular expressions. As there are countless ways of writing many statements, I feel that I would have to spend a lot of time writing, rewriting and fine-tuning regular expressions for this to work.

The second option was using compiler-tools like lex and yacc. This is the way real world compilers like the GNU C++ compiler is made and would involve a lot of C coding and writing yacc specification files. A part of the point of the project and the wishes of my guidance counsellors was that this program should be written as a Python application analyzing Python code.

So I was left with the third option; Python is equipped with a `compiler`-package that analyzes its own code. This outputs an Abstract Syntax Tree (AST) which in turn can be traversed and the nodes analyzed. After playing

a bit with the code and trying out a few programs, I decided to base the code generator on the AST generated by the `compiler` module in Python.

1.2 Project scope

I now had to decide how far I was going to take this. Creating a completely general compiler that could handle every aspect of the Python language would be far too ambitious, so realistic goals had to be set. I wanted to create a program with a framework that makes it easy to expand the program further after the project was initially done. The program would of course have to handle all the normal Python constructs like functions, classes, loops, and so on. In addition, I wanted it to be easy for other developers to expand the range of programs the compiler supports, by adding support for more and more Python modules without them having to get overly familiar with the entire compiler program.

A graphics module should also be implemented to create applications able to run on the Sharp Zaurus SL-5500.

Here follows a brief summary of the rest of the chapters of this document.

Chapter 2 - Background

The background of this assignment is discussed. Why is this program written? Is there a demand for it?

This chapter also contains an introduction to the different technologies used, software as well as hardware. The background of the different languages and the important packages are discussed as well as the hardware the software should support.

Chapter 3 - Code Translation

This chapter contains an explanation of the differences between the Python programming language and C++. This will hopefully shed some light on why it is difficult to translate the code directly and the importance of having good mechanisms for it.

A big part of the program is converting the graphics of a Python application to Qt. The difficulties regarding this are also addressed.

Chapter 4 - The product

This chapter contains an explanation of what the program is and how to use it. The rough layout of the program is explained and how it works together with other applications.

Chapter 5 - Programming

This chapter contains a detailed description of the inner workings of the program. It explains how the code is structured, what classes it contains, and why it is done the way it is. The program also makes use of a number of files and packages that need to be explained here. There is an explanation of what kind

of applications this program is developed for and why. There are also limitations of what kind of Python code the program will accept. Some mechanisms (mostly legal mechanisms that are regarded as bad programming convention) will not be compiled into working C++ code.

There is also a list of C++ packages used by the program. This is C++ implementations of commonly used Python functions and packages.

A section explaining how to make more of these packages to expand the usability of the program is also here.

After reading this chapter you should have a complete understanding of the program and be able to develop it further.

Chapter 6 - Conclusion

In this chapter I share some final thoughts about the project, and list a few other projects that have attempted to create similar programs.

This chapter also contains a summary of the work I have done and what I have accomplished. I will talk about how the furure work will be and how this program should be expanded.

Appendices

The largest parts of this text are the appendices. Appendix A is an installation guide for the program itself and the packages needed. Appendix B is the configuration file of the project. In appendix C, I have included complete transcripts of two test programs that demonstrate the translation of the code. Lastly, appendices D and E contain complete documentation of the Python and C++ code respectively.

These appendices, especially D and E, are fairy large and can be a bit hard to read. Appendices D and E are included with the intention of giving an overview of the different parts of the code of this project.

Chapter 2

Background

2.1 Project background

Programming native code directly for handheld devices can be tedious. Creating a small effective application can be very complicated because the platform on a handheld device often requires a lot of special attention compared to that of a desktop computer. The program will often have to be split into very specific files with very specific structuring and code conventions. This can deter a lot of developers from creating applications for these devices. Having effective code generators that take care of all the "dirty work" and enable the developer to write an effective application in Python without having to worry about all the eccentric requirements of the specific handheld unit, makes software development fun again instead of time consuming and frustrating.

2.2 Sharp Zaurus SL-5500

The Sharp Zaurus SL-5500 [13] is running the Linux 2.4 Embedix operating system with the Qtopia Palmtop Environment. This makes the native language of the PDA, Qt/C++. This is the first handheld unit to incorporate this platform. A big selling point for this PDA is that it is based on an open source Linux platform with all the benefits this has for the availability of applications, simplicity of software development, and the general interest in the open source programming community for developing programs for the device.

2.3 Qt

Qt is a platform independent graphics library written for C++. The purpose of this library is to be able to create single source applications than can compile on multiple platforms such as Windows, Linux, Unix, Mac OS, and embedded Linux. The latter being the main reason it is used in this project. The strength of Qt is that it fills a void in C++, a robust, portable graphics library. See [3].

2.4 Qtopia

Qt is especially attractive because of its compatibility with Trolltech's own platform for handheld units, Qtopia. Qtopia is being used on many PDAs and Smart-phones based on the Linux platform. The Sharp Zaurus SL-5500 I have been using in this project is running Qtopia v 1.5.0. One of the challenges will be to customize an application to run on this PDA.

2.5 Python

Writing small programs in Python [11], [5] that have a lot of functionality can be very satisfying for a developer. The language enables developers to write short, readable code that is easy to maintain. In contrast, C/C++ programs are much more verbose, take much more time to implement, and can be harder to interpret by others.

C++ however is generally way faster than Python in then execution time of programs. Python may be very easy and fast when developing applications, but if speed is of the essence, executing programs with the Python interpreter is generally not a good idea. There are many people who have compared the execution speed of different languages, and Python often gets placed on the bottom half of this list of fast programming languages. Lutz Prechelt has written a comparison ("An empirical comparison of C, C++, Java, Perl, Python, Rexx and Tcl" [6]). A benchmark can also be found here: "http://www.flat222.org/mac/bench/" [9]. This is in itself a good reason to want a translator or a compiler that can take a Python application and create native C++ code that can in turn be compiled into fast and efficient binary code. I have timed a Python program making a mathematical operation and compared it to the execution time of a C++ program making the same calculation. This will be presented in section 4.3.2

Chapter 3

Code Translation

The goal of this project will be to translate or compile Python applications into Qt/C++ applications. These languages are very different in many areas: the structure of programs, how variables are handled, scope, and more. When translating between the languages, these things need to be addressed. This chapter will explain how the languages differ with regards the areas mentioned above.

3.1 Variables

Variables are handled differently in C++ and Python. The main difference is that variables in Python are not type declared explicitly. A variable here is actually a `PyObject` and can be of any type. It can also change type when needed. C++ requires that the programmer explicitly tells the program what type the variable will be so the appropriate amount of memory can be allocated. I will go into details about the differences in the next sections.

3.1.1 Typing

Strong/weak

A discussion often seen on many newsgroups is whether Python is a strongly or weakly typed language. In a strongly typed language, a variable of a certain type cannot act as a variable of another type without an explicit conversion. This gives the programmer a bit more security when writing, but it also lessens flexibility. An example of such a language is C++:

```
int main() {
  int    value1 = 2;
  char*  value2 = "2";
  int    result = value1 + value2;
}
```

This code will not compile. This is because the + operator doesn't know how to add these variables together. It doesn't make sense to add a string and a number together. A weakly typed language like Perl, has no problem adding

these:

```
#!/usr/bin/perl
$value1 = 2;
$value2 = "2";
$result = $value1 + $value2;                                $
```

The above code will run and `$result` will contain the value 4 afterwards. Perl will try to look at the string and figure out what kind of number it represents and perform the arithmetic operation. It doesn't care that `$value2` used to be a string. This is an implicit conversion of the type of the variable. Since the variable is not bound by its type, the language is weakly typed.

Python acts a little differently. Look at the Python code below:

```
#!/usr/bin/python
value1 = 2
value2 = "2"
result = value1 + value2
```

This program will compile and run, but will throw an Error during runtime. Python will accept the code, compile the program and hope that everything will work out. It will not, however, implicitly convert the type like Perl would and throws an Error instead when the operation proves impossible. I feel therefore that Python is strongly typed.

It should be mentioned that there are a lot of disagreement about this point. Some feel that the term Strongly/weakly typed has not been defined well enough to make any absolute statement about a language being either strong or weak. C++ is mostly regarded as strongly typed, but it can sometimes be weaker that Python because of type casting and pointer arithmetic. I will not go further into this discussion here, but this site [4] should be of interest to programmers.

Static/dynamic

There is, on the other hand, no doubt that Python has dynamically bound variables. If the programmer explicitly tells the variable to change type, it will comply. Unlike C++ that has statically bound variables, this is perfectly legal code:

```
value1  = 2
value1  = "2"
```

There is no type declaration of the variables and they can change type as long as this is done explicitly. This property is especially helpful when it comes to argument lists of functions. In a statically typed language the argument list specifies the type of the arguments, but in a dynamically typed language no such declaration is necessary. This enables a function to be able to receive a bunch of different arguments without having to redefine the function. This is made clear in the examples below. This is legal Python code:

```
def myFunction(x):
  print x
myFunction("2")
myFunction(2)
```

To be able to call this function like that in a statically typed language as C++, the function would have to be overloaded like in the following example:

```
void myFunction(char* x) {
  cout << x << endl;
}
void myFunction(int x) {
  cout << x << endl;
}
myFunction("2");
myFunction(2);
```

In a statically typed language, the programmer has to redefine the function and explicitly state what type the arguments should have.

Solution?

This difference in behavior complicates the process of translating the code. Python uses dynamically bound, strongly typed variables and C++ uses statically bound, strongly typed variables. Python does not type declare its variables when they are first used, but a kind of declaration is necessary. To be able to correctly type all the variables, the compiler will have to keep track of the variables and what type they have at all times. If a variable changes type, which is legal in Python, but not C++, another variable has to be made with the type needed and equal scope as the original. Look at this code:

```
a = 5
a = a * a
a = "HEI"
print a
```

This can not be directly translated to C++ without running into typing problems with the `a` variable. To achieve the same, the code will look like this:

```
int a;
char* ab;
a = 5;
a = a*a;
ab = "HEI";
cout << ab << endl;
```

The variable `a` is replaced by `ab`. `ab` is declared at the same place as `a` to ensure the same scope. In the rest of the program, whenever `a` is encountered

it should be replaced by `ab`. The *function problem* is a bit more complicated. There has to be one function for each combination of types in the argument list. If the function is long and complicated, it will be a huge waste of space to just copy the entire function with just different types of variables. It would also probably become very complicated in large functions.

Another way is to wrap the function in some small functions with the single purpose of converting the variable types. This also solves the problem of who should decide the type of the argument: the function itself, or the types of the variables used to call the function? This way, the function will independently decide the types of its arguments, and if the function is being called with a different set of variable-types, another function will be spawned that expects the argument-types given and attempts to convert them into whatever the original functions expect. This is made clear by these examples. First the Python-code:

```
def myFunction(x):
  a = str(x) + ".14"
  print a
myFunction(3)
myFunction("3")
```

This will print `3.14` in both the calls to the `my Function` function. In C++, this will be the code:

```
void myFunction(string x) {
  string a;
  a = str(x) + ".14";
  cout << a << endl;
}
void myFunction(int x) {
  ostringstream oss;
  oss << x;
  string a = oss.str();
  myFunction(a);
}
myFunction(3);
myFunction("3");
```

This will compile and run and it will give the same result as the Python program above. The `str(x)`-function is a Python function that makes sure that whatever `x` is, it is treated as a string. I have made C++ implementation of this function as well as others so they can be used in the same way in the Qt/C++ program. I will explain more about this in section 5.2.5

3.1.2 Strings

Working with strings in C++ can be very frustrating. A string here is nothing more that an array of characters and it is very easy to make overflow mistakes creating unexpected unwanted effects. When concatenating two strings for instance, one has to allocate memory and create a larger string to copy the two strings into. In Python, working with strings is very much simpler for the

programmer. Here string concatenation is done simply with the '+' operator. Therefore I use the package `std::string` in this project rather than char* when working with strings. This complex type supports the normal string manipulation methods used in Python like string concatenation with the '+' operator. This makes the translation of strings a lot easier.

3.1.3 The Variable type

In Python programs there are a number of times where there is no way of determining what type the variable should have. For instance, in lists and dictionaries the elements can be of any type and can change. In these situations the program creates a variable with the datatype `Variable`. This is a class and can represent any kind of variable and can also change type when required. The class consists of an attribute specifying what type the variable currently has and multiple attributes which contains the value of the variable. If the type is a basic datatype like integer or double, it is stored in `int` and `double` variables. More complex datatypes like `PyDict`, `PyList` (explained in the following sections), or any other objects are stored in a `void*` variable to make it able to contain any type.

Whenever the compiler is unable to determine what type a variable has, it will be given the type `Variable`. Whenever this variable is used, it will out of the context figure out what type it is expected to have. This is often the case when dealing with lists. When extracting an element from a list without specifying what it is expected to have, there is no way the compiler can know what type to extract the element as. For instance when the element is being used in a `print` statement it is expected to be textual and will therefore call on the method `toString()`. This returns a textual representation of the variable whatever kind of value it contains.

3.2 Arrays/Lists

Arrays work in different ways in Python and C++. In C++ an array is type specific. This means that when an array is created, you have to decide what type of elements the array should store. No other type will be allowed in this array. You will also have to decide the maximum number of elements the array will allow. These restrictions do not apply to Lists as they are called in Python. Here a List is not typed; any type of object is allowed in any List no matter what kind of elements are there to begin with. And there is no need to specify how many elements to make room for.

Because of this difference in behavior, C++ arrays can not be used to represent a List in a Python program. To solve this, references to Lists in a Python program will be translated to calls to various functions in C++. I will illustrate by the following example.

```
a = ["WORLD",5]
a.append("HI")
print a[1]
```

This will print `5`. The corresponding C++ program will look like this:

```
PyList a;
PyList temp;
temp.append("World");
temp.append(5);
a = temp;
a.append("HI");
std::cout << a.getString(1) << std::endl;
```

As shown, when a list is encountered (`["WORLD",5]`), a temporary variable is spawned and the elements appended to it. This list will then be passed as an argument instead of `["WORLD",5]`. This enables lists like this to be used as arguments to functions just like in Python.

The program above will also print `5`. The PyList is a class which emulates the behavior of a python-list. When initializing the list, the program adds the items to the list with the `append` method. The PyList class includes many functions that make it behave like a List. The elements are stored in a `<vector>` that allows any type of element to emulate the flexibility of the Python List. The elements stored in the vector are instances of the `Variable` class described above. This way, the elements can be of any type. For instance a dictionary (described in the next section) or another list to create a multidimensional list.

3.3 Dictionaries

Python also has a construct called Dictionary, or `Dict` for short. These are similar to Lists in that they store a number of elements that can be of any type. The difference is that elements in a Dictionary are extracted by the use of a key of some sort instead of an integer as in a List. This *key* can be of any immutable type like strings and tuples. The most common usage is of course the use of strings as keys. Like this:

```
a = {"Fot" : "Foot",
     "Kne" : "Knee"}
print a["Fot"]
```

This will produce the output `Foot`. This usage is probably why it is called a Dictionary in the first place. The key is as mentioned not limited to a string, but the most common usage is strings. Therefore I have decided to reduce the complexity by restricting the usage of dictionaries to only allow strings as keys.

To emulate the behavior of the dictionary a `PyDict` datatype is used. This behaves very much like the `PyList` type above. It is implemented with the `<map>` functionality from the C++ standard library and uses `string` as the key and `variable` as the value. This way, the value can be of type `PyDict`, `PyList` or any other type of element.

The above example will translate to this C++ code:

```
PyDict a;
PyDict dict_temp;
dict_temp.setValue("Fot", "Foot");
dict_temp.setValue("Kne", "Knee");
a = dict_temp;
std::cout << a.getString("Fot") << std::endl;
```

The same mechanisms as used in lists are used here; a temporary variable is spawned and the initialization values are added to it, then it is passed on in the original dictionary's stead. This program will also of course produce the output `Foot`.

3.4 Keyword arguments

Keywords as arguments to functions in Python is a mechanism not present in C++. It allows for an arbitrary number of arguments to be passed to a function or a method and the programmer will specify what kind of arguments is passed, rather than relying on the order in which they are sent which is the conventional way of passing arguments in C++. The usage in Python is like this:

```
myMethod(arg = 8, anotherArg = 3);
```

This is a powerful mechanism because it allows for a variable number of arguments and they can be specified in any order. This can not be directly translated to C++.

To emulate this behavior, whenever keywords are encountered like in the previous example, an instance of the class `Keyword` is spawned. This class behaves very much like a dictionary and "loads up" with all the keywords specified in an argument list. The instance of the class is then passed as an argument in the keywords' place. This approach is very much akin to the approach used in the `PyList` and `PyDict` classes above. The target C++ code will clarify:

```
Keyword key_temp;
key_temp.add("arg", 8);
key_temp.add("anotherarg",3);
myMethod(key_temp);
```

The method `myMethod` will need to take a reference to a `Keyword` class as argument: `void myMethod(Keyword& kw)`. It also needs to contain code to handle the keywords that is sent. This mechanism is used to a great extent in my implementation of the graphics library which is discussed later in this document.

3.5 Python's "self"-thingy

The first argument of Python functions that are contained within a class, have a special meaning in Python. It is a reference to the class itself and is used to access variables that are declared on the upper block level. This should, by convention, always be named `self` to ensure compatibility with other Python-packages. In C++, there is no such explicit first argument, but it is still there behind the scenes. The `this` variable is, in C++, a keyword representing a pointer to the class itself and, as the `self` variable in Python, it is used to explicitly access variables on class-level. These mechanisms are basically the same with just a small difference in usage. To make sure that the translation will maintain the same functionality, the first argument in a Python function declared within a class is simply stripped when translating it to C++. Then a variable with the same name, typically `self` as mentioned above, is created that point to `this`. Now `self` can be used to refer to the class itself or access its class-level variables in the same way as in Python. Here is a code segment to illustrate:

```
class myClass:
  def myFunc(self):
    pass
```

This translates to:

```
class myClass {
public:
  myClass();
  void myFunc();
}; // end of class myClass

void myClass::myFunc() {
  myClass *self = this;
} // end of function myFunc

int main(int argc, char* argv[]) {
  return 0;
} // end of function main
```

As shown, the original `self`-argument is stripped and made into a variable which points to `this`. When `self` is used further on in the function, the target C++ code will point to the `self` pointer for the same functionality.

Chapter 4

The product

The program I have developed is a translator or a compiler to convert Python applications to Qt/C++. The program, called `qGen` takes a Python application as input and generates Qt/C++ code able to compile and run on a machine running the Qtopia platform.

The application itself is written in Python. The Python API contains a well suited package to analyze its own code. The application works by taking a Python application as input and generates C++ code. The code will be divided into suitable .cpp and .h files. Quite a few pre-made Qt/C++ packages will also be included. Most programs use Python functions, built into the language, which are not available in C++. A small library of C++ implementation of often used Python specific functions is included to the target code.

Other library files often used in Python like the `Tkinter`-package and the `math`-package also have C++ implementations in the library. `Tkinter` is the graphics module in Python that I have concentrated on creating support for. This will be explained more in section 5.2.5

The installation of the `qGen` program consists simply of decompressing and unpacking the `tar.gz` archive `qGen-1.0.0.tar.gz`, and setting the environment variable. Se appendix A.1 for an installation guide and an overview of the content of the archive.

All the finished .cpp and .h files generated are packed into a file structure containing all control files, icons, libraries, make-files and source code needed to build the application to run on the Qtopia platform. A cross-compiler developed by Trolltech will compile all files into an .ipkg file that can be transferred and installed on the Qtopia platform. The building of the file structure and creation of the control files and other essential files are done by some simple scripts. These scripts also handle the compilation and building of the finished .ipkg file.

Since the source code being generated is platform independent, it is of course possible to build the application for any desktop system supported by Qt (which is most). By command line arguments to the application, the code will be built for the specified platform. I have made scripts to build the application on Unix/Linux in addition to the Qtopia platform.

For the compilation of the target Qt/C++ code to work, some setup is required. To compile on and for a desktop computer using Qt, the Qt/X11 package has to be installed. To compile for the Qtopia platform, the Qtopia package and Qt/Embedded need to be installed. This package contains a virtual

frame buffer to emulate the Qtopia platform on a desktop computer and is very useful during development and testing. If the target code should be compiled for running on a handheld device, like the Zaurus, a cross-compiler has to be installed. Installation and setup of these packages can be very tedious and time consuming. Just downloading and compiling the packages needed takes many hours, and setting up the environment with all environment variables pointing to all the different libraries for different platform is a very long and frustrating process. Many web pages try to give a quick guide to the setup of these packages, but they are all 'ad hoc' and rather incomplete.

The best way of going about the installation of these packages, is to download the latest packages required from the Trolltech web page [12] on `http://www.trolltech.com/download/`. I will not go into a lengthy explanation of the setup here, but see appendix A.2 for a guide to the installation of these packages.

4.1 Supported constructs

The program currently supports a number of "normal" operations. It handles the creation of classes and instantiation, it also handles for- and while-loops and if-else blocks. It also handles variables of different types and the change of types. You should however be a little careful when dealing with the different variables. If the program is unable to figure out the type of a variable, it will be given a general type. This often happens in for-loops and when dealing with lists and dictionaries. When dealing with these constructs it would be a good idea to tell the compiler what type the variable has. Take this example:

```
for i in methodThatReturnsAPyList():
  doSomeThingWith(i)
  list.append(construct[i])
```

This can be difficult for the compiler. Just by looking at this it is for instance hard to tell if `construct` is a Dictionary or a List. In cases like this you should help the compiler out by replacing `i` with `int(i)` if you expect `i` to be a integer or `str(i)` if you expect `i` to be a string. In most cases the compiler will be able to choose correctly be the context, but it cases like the one above, a little help would be worthwhile.

Some mathematic capabilities are also implemented. The program supports adding lists and multiplying with integers. If the math module is imported these functions are available:

acos	atan2	exp	fmod	pow	sqrt
asin	ceil	fabs	log	sin	tan
atan	cos	floor	log10	sinh	tanh

There is also (limited) support for the `sys` module. This can be used to retrieve command-line arguments.

For graphics, the Tkinter module is supported, but limited to the classes listed below. Always use the `grid`-layout when placing the widgets. Programs

that use `pack`-layout will compile and run, but the widgets will most likely be badly misplaced.

Tkinter	Qt	Tkinter	Qt
Tk	QApplication	Label	QLabel
Frame	QFrame	Entry	QLineEdit
Button	QPushButton	RadioButton	QRadioButton
StringVar	QString		

Remember that when importing these modules, always use the `import` statement, not `from`. The module must be prefixed whenever a class or function is being called from it.

4.2 Usage

To compile a Python application into working binary code, we go through a two-step process. First the Python program `qGen` is called with the Python source code as argument. We also need to specify a few arguments to the application. The arguments are:

```
-stdout             Prints the target Qt/C++ code to the standard output.
-o <dir>            Writes the target Qt/Q++ code to files in the directory <dir>.
                    When this option is specified, additional scripts and
                    makefiles are also created in this directory.
-p <platform>       Specifies the platform the target code should be compiled on.
                    This should only be used in conjunction with the -o option to
                    create files in the directory to compile the project for the
                    specified platform. <platform> can be any of the following:
                      X11        Normal desktop computer running X. (default)
                      Embedded   Using the embedded version of Qt to create
                                 code to run on an embedded device.
                      Qtopia     Using the Qtopia library to create binary
                                 code runnable on a Qtopia system. When using
                                 this option the program will run on the Qtopia
                                 virtual framebuffer.
                      Sharp      Uses the cross-compiler and the Sharp library
                                 to create a program able to run on the Sharp
                                 Zaurus PDA.
-name <name>        Specifies the name of the program. This will be the caption
                    of the icon. Default is the filename of the main source file.
-desc <description> A textual description of the program. Default is none.
-icon <filename>    The icon that should be used for the program.
-exec <filename>    The name of the target executable. Default is the filename of
                    the main source file without extension.
--help  or  -h      Print out this usage information.
--qtX11_path        The path to the Qt installation. Default /usr/lib/qt-3.1
--qtEmbedded_path   The path to QtEmbedded. Default /opt/qtEmbedded
--qtopia_path       The path to Qtopia. Default /opt/Qtopia
--crosscompile_path The path to the sharp crosscompiler. Default /opt/Embedix
```

This script requires the `$QGENPATH` environment variable to be set. This should be set to the directory where the qGen application is installed. For instance `/opt/qGen`. The `$QGENPATH/bin` directory should also be added to `$PATH` or a symlink to the file `$QGENPATH/bin/qGen` should be put in a `bin` directory that is already in path.

The program will create the directory specified with the -o option and put all target files there. The second step is to build the executable program from

the Qt/C++ code. The program in step 1 will create a `configure` script in the target directory. This script creates all project files and link libraries that is needed and creates a `Makefile`. The `configure` script also sets the environment variables needed to compile the program for the target platform.

After this, run `make` to build the program. The executable will be put under the `bin` directory under the target directory. To install the program on a Sharp Zaurus, run the script `ipkgmake`. The target file can be transferred and installed on the Zaurus.

Here is an example on building a program for the Zaurus from a source Python program in the `myProgram.py` file in the current working directory:

```
> qGen myProgram.py -o myProgram -p Sharp -name myProgram \
  -desc "This is a test" -icon myProgram.ico
```

This will create a directory in the current working directory called `output`. All files needed to build the finished program and installation package will be put here.

```
> cd myProgram
> ls

apps bin configure ipkgmake myProgram.control myProgram.cpp pics

> ./configure
> ls

Makefile bin        ipkgmake           myProgram.cpp
apps     configure  myProgram.control  myProgram.pro

> make
> ./ipkgmake
```

The `make` command compiles the program according to the rules created by the `configure` command. The `configure` script is built using the options specified when the `qGen` script was called. The `ipkgmake` script packs all the finished files into a package which can be transferred and installed on the Sharp Zaurus.

The process is done in these steps to make sure that the user has complete control over what happens, and is also able to make last minute alterations and changes if she desires. This usage, where the process is divided into three steps, can seem a bit complicated for users not familiar with building programs from source on a UNIX platform. I feel that it is a good idea to force the users to gain a little bit of familiarity with the building process, and by doing so, enabling them to make alterations on the way to creating the final product.

4.3 Test programs

I have written two test programs to show the practical use of the `qGen` program. I have tried to incorporate different aspects of Python that the program sup-

ports. The first program is a very simple app to show class instantiation and inheritance. The second is a GUI application that performs a mathematical approximation of the area beneath a graph. In the last example the efficiency of the target application is compared to the source Python application.

4.3.1 Inheritance

The first program is a small application that shows instantiation of classes and inheritance. It contains four classes: One that contains two functions, and the others are three classes that inherits the first. The subclasses are instantiated and the functions of the super class are accessed. The complete Python code of this program can be found in appendix C.1.1.

The `inheritance.py` program simply creates instances of the classes, gives them names, and extracts the names from each class before printing them out to the standard output. When the program is executed, this is the output:

```
> ./inheritance.py
1st 2nd 3rd
```

The following example show how to translate this Python application to C++ and build it using a `configure` script provided with `qGen`.

```
> qGen inheritance.py -o inheritance
Using qGen: /ifi/tyrfing/h09/martinje/hovedfag/python/qGen
Using Qt/X11 library: /usr/lib/qt-3.1
Creating directory: inheritance
Processing file: inheritance.py
......
Done processing file.
Writing target code to files.
  Writing file: inheritance/inheritance.py.A.h
  Writing file: inheritance/inheritance.py.C.h
  Writing file: inheritance/inheritance.py.B.h
  Writing file: inheritance/inheritance.py.Super.h
  Writing file: inheritance/inheritance.py.main.cpp
Creating project file.
Creating configure script.
Done.

Go to directory 'inheritance' and run './configure' and 'make'.
```

```
> cd inheritance
./inheritance

> ./configure
Running qmake to create Makefile
Configuration done. Run 'make' to build program.

> make
qmake -o temp_Makefile inheritance.pro
g++ -c -pipe -Wall -W -O2 -g -pipe -march=i386 -mcpu=i686
    -fno-use-cxa-atexit -fno-exceptions  -DQT_NO_DEBUG
    -I/usr/lib/qt-3.1/mkspecs/default -I. -I. -I../../lib
    -I/usr/lib/qt-3.1/include -o inheritance.py.main.o
    inheritance.py.main.cpp
inheritance.py.main.cpp: In function `int main(int, char**)':
inheritance.py.main.cpp:29: warning: unused parameter `int argc'
inheritance.py.main.cpp:29: warning: unused parameter `char**argv'
test -d bin/ || mkdir -p bin/
g++  -o bin/inheritance inheritance.py.main.o   -L/usr/lib/qt-3.1/lib
     -L/usr/X11R6/lib -lqt-mt -lXext -lX11 -lm

> bin/inheritance
1st 2nd 3rd
```

The warning messages from the compiler simply states that the arguments to the main method are not used. When building larger projects, more warnings like these should be expected, but not feared. The `self` argument of Python functions is a recurring source for complaints by the C++ compiler, as these arguments often stand unused. These warnings could of course be suppressed, but they can be of help to programmers who want manually to optimize the code generated by the `qGen` program.

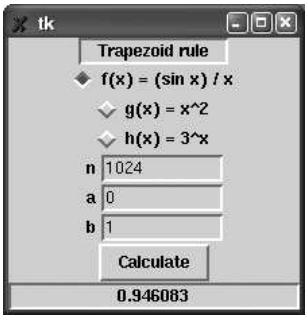
The last command above prints the same output as the Python program. As shown by the output from `qGen`, the target C++ code is split into the five files `inheritance.py.Super.h`, `inheritance.py.A.h`, `inheritance.py.B.h`, `inheritance.py.C.h`, and `inheritance.py.main.cpp` and put in the directory `inheritance`. A `configure` script and a project file is also placed in this directory. Appendix C.1.1 shows the combined C++ code for the classes `A`, `B`, and `C`. The code will actually be split into three files, but it is condensed into one in the transcript. The classes all inherit the `Super` class. A transcript of the `inheritance.py.Super.h` file can be found in appendix C.1.1.

The code not contained in a class is put in the file `inheritance.py.main.cpp`. This contains the `main()` method that starts the program. A transcript of this file is in appendix C.1.1. The header files of the application are included in this file. The program makes sure that the file containing the superclass is included before the files with the classes that inherits the superclass.

4.3.2 Trapezoid Rule

The second program is a GUI program. It calculates the area beneath a graph using the Trapezoid Rule approximation method. It displays three radio buttons where the user can choose between three different graphs. The user also determines the boundaries of the graph by specifying `a` and `b` and the number of intervals (`n`). The higher the number of intervals is, the closer the approximation will be to the actual number. Setting a high number will of course slow the program down.

The Python program looks like this:



The usage for building the program are the same as for the example above:

```
> qGen trapezoidrule.py -o trapezoidrule
Using qGen: /ifi/tyrfing/h09/martinje/hovedfag/python/qGen
Using Qt/X11 library: /usr/lib/qt-3.1
Creating directory: trapezoidrule
Processing file: trapezoidrule.py
.............................................
Done processing file.
Writing target code to files.
  Writing file: trapezoidrule/trapezoidrule.py.Functions.h
  Writing file: trapezoidrule/trapezoidrule.py.main.cpp
Creating project file.
Creating configure script.
Done.

Go to directory 'trapezoidrule' and run './configure' and 'make'.

> cd trapezoidrule
./trapezoidrule

> ./configure
Running qmake to create Makefile
Configuration done. Run 'make' to build program.

> make
```

The Python code consists of a class with three functions. These are the functions of the graphs the user can choose from. The Python code for this

class can be found in appendix C.2.1. The functions make use of the `math` module to calculate the graph. The three differing functions simply take the `x` coordinate of a point as input and return the `y` coordinate.

The graphs provided are:

$$f(x) = \frac{sin\ x}{x}$$
$$g(x) = x^2$$
$$h(x) = 3^x$$

The class with the functions is put in its own file in the target code. The class and all its functions are first defined abstract, then all functions are implemented including the constructor (which is empty in this example). Se appendix C.2.1 for a transcript of the file `trapezoidrule.py.Function.h`. The `self` argument is not used in this example, but the argument is extracted from the argument list and created below as a pointer to the class itself.

The rest of the file is not contained in a class and collected in the file `trapezoidrule.py.main.cpp`. This consists of three functions. The first is `trapezoid`. This takes the range, `a` and `b`, and the number of intervals, `n`, as input and outputs the area below the graph. The function uses the Trapezoid Rule approximation method to calculate the result. When the user makes a choice in the group of buttons, the `activeFunc` element is updated. The `trapezoid` method uses the value of `activeFunc` to determine which function (`f(x)`, `g(x)`, or `h(x)`) to use.

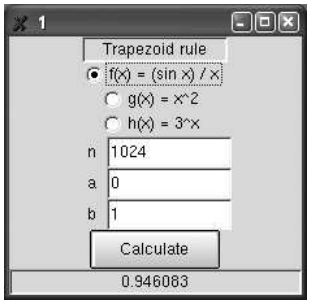
The second function is `execute`. This is the function connected to the button. This reads the values the user has specified in the input-fields and calls upon the `trapezoid` method with these arguments. Appendix C.2.1 shows the C++ code for the functions `trapezoid` and `execute`. In the header of this file, all GUI components that is used in the program is being declared. All the variables that are used on *class level* are declared in the head of the file. This is because the scope rules are a bit different i C++ and Python, so the variables can not be declared "on the fly".

First, all the functions are declare abstract, then the implementation. Notice that every time a built-in function like `range` is used, it is prefixed by the namespace `builtin::`. The for-loops use a PyList to iterate over to emulate the effect in Python. The `initialize()`-, `hasMore()`-, and `nextElement()`-methods are methods implemented in `PyList`.

The rest of the target code are the creation of the GUI components and the call to the `mainloop` of `Tk` which starts the program. This code is placed in the appendices C.2.1 and C.2.1. The code can seem a tiny bit hard to read because of all the temporary lists with random names. These lists are the keyword arguments that are sent to miscellaneous functions. As mentioned in section 3.4, since there are no keyword arguments in C++, these arguments are put in an instance of the `Keyword` class. This instance is sent as argument instead of the keywords themselves.

Aside from the clutter that the random names of the temporary arguments present, the target code should be fairly readable. All the extra bit of code that is needed to make C++ behave like Python, are put in header files that are included to the program.

Here is a screen shot of the Qt/C++ application:



Execution time

This program can be used to make an interesting experiment. I removed all the GUI components from the Python program and timed the execution using the arguments `a=0`, `b=1`, and `n=5000000`. Then I translated this program with `qGen` and ran it using the same arguments. This is the result:

```
> time python trapezoidrule.py
0.94608328622

real    0m25.530s
user    0m24.760s
sys     0m0.400s

> time trapezoidrule
0.946083

real    0m6.367s
user    0m5.580s
sys     0m0.720s
```

The target program is more than 4 times as fast as the Python program. The programs are executed on the same machine with as little time between the executions as I could manage. The comparison has been made many times, always with very similar results to the one shown. This comparison suggests that translator programs like this can be very useful when writing programs that perform large computations, and when efficiency and speed of the program is important.

Chapter 5

Programming

5.1 Introduction

The program consists of a Python application and a collection of C++ library-files. The main program is written in Python. It analyzes the Python code it receives as input, and uses a small number of other Python classes to generate the target code. The library-files contain namespaces with classes and functions that are needed by the target code to function as the original Python program. In addition, the program also requires an XML configuration file (see appendix B). This file specifies which Python packages the program supports and where the Qt/C++ implementation is.

5.2 Program structure

This section and the rest of this chapter contain details about the inner workings of the program. This is intended for people who wish to add support for more packages or want to develop the program further. It is recommended that you read the rest of this chapter before you dig into the programming code. Some parts can be a bit difficult to explain (and even harder to understand) in so many words. I will try to give many relevant examples to clarify in the following sections.

5.2.1 Code structure

The main program behaves like a normal compiler by reading the source code of the program to be compiled, building an abstract syntax tree (AST), and then traverse through the tree analyzing the nodes. A separate class, called `CodeGenerator`, will keep track of all the variables, functions and classes, and other information that may be needed later on. While the main program traverses through the code trying to figure out what to do with the nodes, it will constantly check with the `CodeGenerator` for information previously gathered.

When the main program determines what to do, it will send the code to the CodeGenerator and it will be stored in a class hierarchy. The CodeGenerator class has the subclasses `Class` and `Function`. These, in addition to keeping

track of virtually everything that is going on when traversing the AST, also keep all the target code.

When the traversing is finished, the program will normally store the target code in a number of files. What the program actually does when done traversing is configurable by arguments to the program. Se chapter 4.2 for details.

The target code will then depend on C++ library files which contain C++ implementation of Python functions and classes. For example the Python `range()`, `str()`, or `int()` functions are built-in in Python. The program will treat these functions as any other function and the target code will call on corresponding C++ functions. These are contained in a namespace called `builtin`. Other namespaces contains more functions needed by the target program.

5.2.2 Main program

The main program is purely functional without any classes. It starts by reading the entire source code and building an abstract syntax tree (AST). Then it traverses through the tree by analyzing the nodes one by one and calling upon the corresponding functions. For instance, if the node is a class node, it will call on a method called `nodeClass` with the node itself as parameter. The node will have quite a few children; one will contain the name of the class, another will contain the classes this class inherits, if any, and so on. The `nodeClass` function will know how to treat each node and call the appropriate function. The program will all the time send all extracted information to the `CodeGenerator` for storage. This can be information like the name of the current function and class, the type of the last variable encountered, and so on. All this information is used by the `codeGenerator` to determine how to treat the code that is sent to it.

The documentation for this program is extracted using `Epydoc` [8]. The documentation can be found in appendix D.1.

The main program also makes use of a few help tools. These are contained in the `Tools` class. The documentation for this class can be found in appendix D.3

5.2.3 CodeGenerator

The `CodeGenerator`class contains all target code and all temporary information the program needs. It has two inner classes; `Class` and `Function`. The `CodeGenerator` contains pointers to all classes as well as pointers to functions that are not contained in a class. `Class` contains pointers to all functions in the class as well as target code that is not contained in a function. The `Function` class contains the target code that should be contained within a function. All classes also keep track of all the variables that are visible and those that should be declared.

Documentation for this class can be found in appendix D.2

5.2.4 Graphical explanation

It is a bit difficult to explain how this works in just words. I will try to clarify by these illustrations:

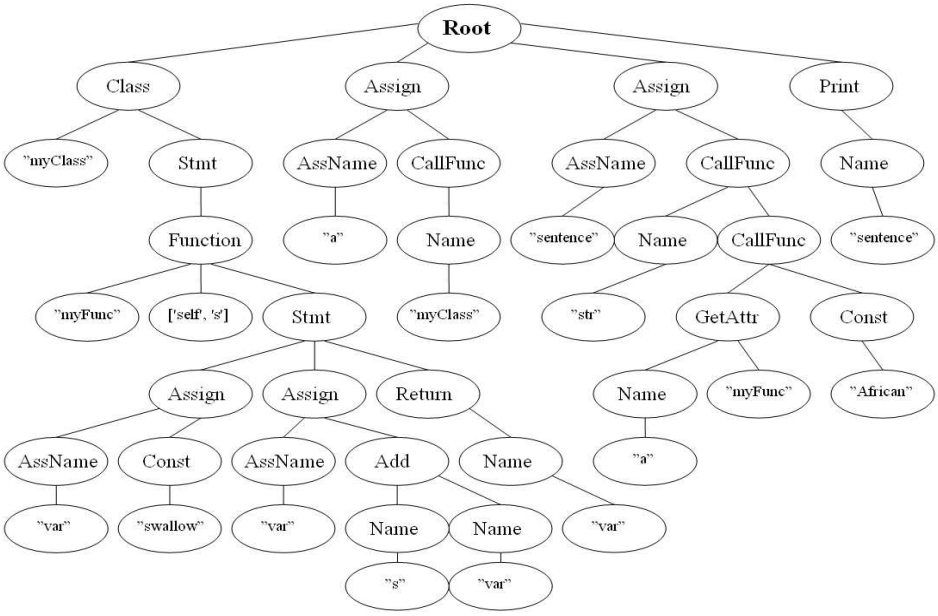


Figure 5.1: A simplified illustration of what the AST generated from the code will look like.

I will illustrate using this Python code:

```
class myClass:
  def myFunc(self, strin):
    var = "swallow"
    var = strin + var
    return var
a = myClass
sentence = str(a.myFunc("African"))
print sentence
```

First the program will read the code and build an AST representing the program. Figure 5.1 shows what the AST generated from this code will look like.

The nodes in the AST shown in figure 5.1 are the essential nodes produced from the program printed above. The tree contains quite a few more nodes, but to save space I have omitted the ones that do not contain information relevant to this example. The Class node for instance has another child which contains the names of the classes this class inherits. Since there are none, that node is empty.

The AST is then traversed in pre-order calling on a method for each node. When the program analyzes its first node, `Class`, it expects it to have a node specifying the name of the Class (myClass) and a node containing the statements that should be executed in the body of the class (it also expects a number of other nodes that is irrelevant to this example). The statement-node, `Stmt`, can have an arbitrary number of nodes, one for each statement of the class' body.

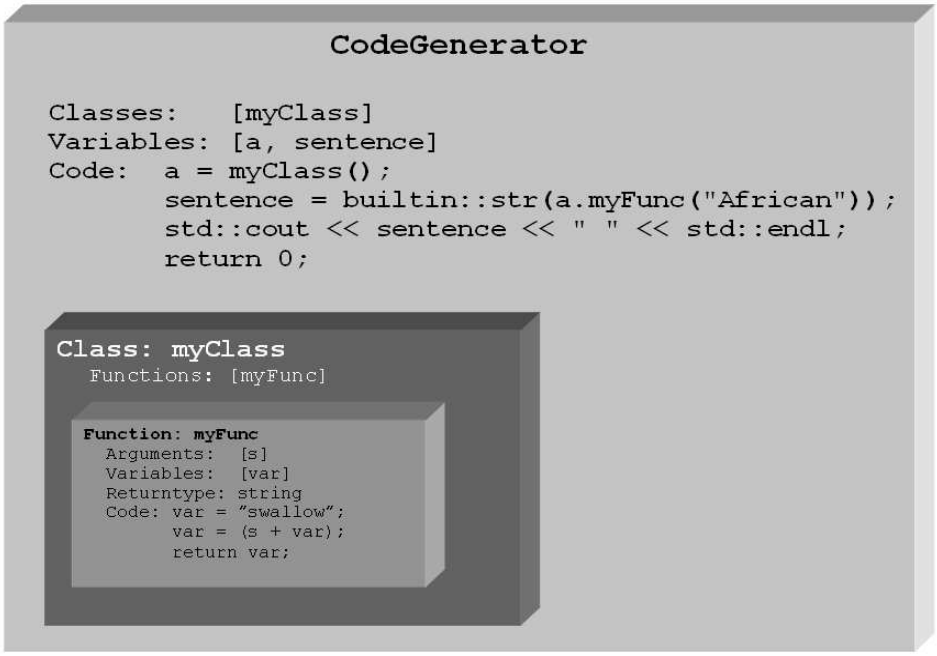


Figure 5.2: A simplified illustration of how the CodeGenerator stores the information.

In this example it only has one, `Function`. This node will, as the `Class` node, expect one node containing its name (myFunc) and another containing the statements in its body. In addition, it will have one child containing the arguments of the function (This node also has more children, like a node specifying the default values, if any, of the arguments, but they are irrelevant to this example and is omitted).

If an AST like this would be built from a program written in a language like C++, you would be able to draw a bit more information out of it. It would for instance contain information about the return type of the function and the types of the arguments the function could expect. This is information that is not needed in a Python program, but essential in a C++ program, so by using the information given by the AST above alone, you can't build a corresponding C++ function.

The `CodeGenerator` will reserve space in its hierarchy for the information needed, and when the program analyzes the function's body, it will keep an eye out for what these variables are used for. When it figures out what type these variables are expected to have, it will fill out the blanks in the `Function` class within the `CodeGenerator`.

Figure 5.2 shows how the class structure is within the `CodeGenerator`. This illustration is also a simplification. The actual object contains more information, like the types of the variables and parameters and the default values, if any, of parameters to functions. The attributes shown are the most relevant. After the AST has been traversed and the `CodeGenerator` classes has been built, the program will traverse the `CodeGenerator` and output files with the target

Qt/C++ code. Each class will first be declared without implementation of the functions to make sure that all classes and functions are declared before they are referenced. Functions that are not contained within a class are also declared abstract before the implementations.

The output will be as follows.

```
class myClass {
  public:
  myClass();
  string myFunc(string strin);
}; // end of class myClass

string myClass::myFunc(string strin) {
  string var;
  myClass *self = this;
  var = " swallow";
  var = (strin + var);
  return var;
} // end of function myFunc

/* Global variables declarations */
myClass a;
string sentence;

/* All functions */
int main(int argc, char* argv[]) {
  a = myClass();
  sentence = builtin::str(a.myFunc("African"));
  std::cout << sentence << " " << std::endl;
  return 0;
}
```

As mentioned earlier, the actual code produced by the program will be split into appropriate files. All class definition and implementation of the methods will be put in separate files. The main method and methods put outside of classes will also get its own file. A number of header files will also be included in each file. I have stripped the code presented in these examples a bit for the sake of readability, but all relevant code is shown.

5.2.5 Additional packages

As mentioned, the target code that is directly generated from the source code will be built very much like the source code. Python code can often achieve very much in just a few statements while C++ uses a more verbose approach which needs a lot more code and specification. In such cases, like generation of lists or for-loops, the target C++ code will often be as short as the Python code, but it will call upon external functions that contain the required C++ code.

These functions are contained in a library of namespaces. For instance, Python often uses `range(...)` to create a list of integers and `str(...)` and `int(...)` to explicitly convert variables from one type to another. In the library, these (and more) functions are contained in a namespace called `builtin`.

All the header files in the library are documented by documentation strings in the code. This is extracted using the `KDOC` tool [2]. All documentation of these classes can be found in appendix E. The documentation consist of many pages and can be a little tough to read through, but hopefully it can give the reader an idea of the code structure and the hierarchy of the classes.

For each package that is supported by this program, there is a corresponding namespace which contains the classes and methods needed. For instance if the source code imports the `math`-package, the target program needs to include a header file containing a math-namespace. This namespace should contain all the methods that the `math`-package of python contains. This example shows this:

```
import math
print math.sin(1)
```

This will result in this target code:

```
#include <iostream>
#include "c_math.h"
int main(int argc, char* argv[]) {
  std::cout << math::sin(1) << std:endl;
  return 0;
}
```

These both print out approximately `0.841`.

This is the way all packages are handled. Every time a package is included from the Python library, a corresponding namespace is included in the target C++ program. This way, it is easy to expand the library with supported packages. All packages and methods supported are configured in an XML-file. This file also specifies the filename of the C++ header file that contains the namespace and which methods that are implemented in this namespace. To add supported packages and methods to the program, the namespace and the methods that are implemented have to be specified in this file. No change in the compiler itself is necessary.

The XML file is built like this:

```
<Packages>
  <Package pname='{...}' cname='{...}' file='{...}>
    <Method>
      <PName>{...}</PName>
      <CName>{...}</CName>
      <Type>{...}</Type>
      <Description>{...}</Description>
    </Method>
  </Package>
</Packages>
```

The file contains one `<Packages>`-tag and this can contain any number of `<Package>`-tags with any number of `<Method>`-tags wrapped inside. The arguments `pname` and `cname` are the Python name of the package and the C++

name of the namespace respectively. The `file` argument is the name of the C++ header file that contains the namespace.

Inside the `<Method>`-tags, `<PName>` and `<CName>`, in the same way as above, represents the Python- and C++ name of the function. `<Type>` is the return type to be used in the C++ target code and `<Description>` is a textual description of the method that isn't used at all by the compiler. It is just provided for the sake of readability.

Standard library

Python has a number of functions that are built into the language in the sense that the programmer does not need to import a package to use it. These are functions like `str(...)`, `int(...)`, `range(...)`, et cetera. These methods are contained in a namespace called `builtin`, which is always included in the target code. This is used in the example in section 5.2.4.

The documentation for the standard library can be found in appendix E.18

Math library

The `math` module of Python consists of a number of methods to perform mathematic operations. These are methods that are interfaced from the `math.h` library of C++, so I am just interfacing it back to my `c_math.h` library. I have interfaced most of the methods in the module, but not all. Below is a list of the supported methods.

acos	atan2	exp	fmod	pow	sqrt
asin	ceil	fabs	log	sin	tan
atan	cos	floor	log10	sinh	tanh

The documentation for this library can be found in appendix E.19.

Graphics library

The most commonly used graphics library in Python is the `Tkinter` package. This is the graphics library I have concentrated upon. It is translated to the Qt library in C++ which is described more in section 2.3. It is implemented in the `c_tkinter.h` library in a namespace called `Tkinter` and consists of the most commonly used classes in Python. The classes contain all methods that the program supports. See the configuration-file in section B for a listing of all supported methods and classes. How to add more methods and classes are described in section 5.3.

All classes in the `Tkinter` namespace that represents a widget in Python's `Tkinter` module will be connected to a Qt widget that functions as close as possible to the Python widget. For instance the `Label` widget in Tkinter is represented by a `QLabel` in Qt and so on. Below is a list of the Tkinter widgets and their corresponding Qt widgets. I mentioned these in section 4.1, but I list it here too for completeness.

Tkinter	Qt	Tkinter	Qt
Tk	QApplication	Label	QLabel
Frame	QFrame	Entry	QLineEdit
Button	QPushButton	RadioButton	QRadioButton
StringVar	QString		

The `Tkinter` packages support two different layout managers: the `pack`- and the `grid`-manager. The `pack` mechanism proved to be very difficult to translate to Qt, since there is no Qt layout manager that behaves in the same manner. Qt does have a `QGridLayout` that behaves more or less the same way as the `grid`-layout in Tkinter. Therefore it is strongly advised to use the `grid` layout exclusively. The `pack` methods will still work, but the result will very likely not be as expected.

The graphics library Qt is interfaced for Python and called `PyQt`. The reason I have decided to use the `Tkinter` package instead of `PyQt` is that `Tkinter` is more common in existing programs. Without going into too much detail, the use of `PyQt` would not make the implementation in Qt/C++ significantly easier. The structure of the `c_tkinter.h` library would be very much the same. It would, however, probably be easier to find widgets in `Qt/C++` that behave the same and have the same properties as the widgets in `PyQt`, but this has not been much of a problem with the `Tkinter` package either.

The documentation for the graphics library can be found in appendix E.3.

5.3 Adding additional packages

Adding more packages to the library is not difficult with this framework. By writing C++ libraries that emulate the same functionality as the Python packages, the compiler can be expanded to support many different programs and packages. By writing the namespaces and specifying the packages and functions in an XML configuration file, support for the package is added without having to add code to the compiler itself.

When writing the namespace, a few conventions need to be taken into account. The framework will look like this:

```
namespace <name> {
  <returntype> <methodname>(<arguments>) {
    ...
    <code>
    ...
    return <something>
  }

  class <classname> {
  public:
    void init();
  }
}
```

The namespace can consist of static methods unrestrained by a class as in

the `math` library mentioned above or it can contain classes or nested namespaces. It also needs a few other specific functions. `init()` should emulate the behavior of the class' constructor and can have arguments.

```
import package
a = package.Class()
```

This Python source code will be translated to this C++ target code:

```
#include "c_package.h";
ns_class::PyClass a;

int main() {
  a.init();
  return 0;
}
```

The `c_package.h` file will have to look something like this:

```
namespace ns_class {
  class PyClass {
  public:
    void init();
  }
}
```

The methods `init()` should of course be implemented to do whatever the class constructor does.

The XML-file will look like this:

```
<Package pname='package' cname='ns_class' file='c_package.h'>
  <Class>
    <PName>Class</PName>
    <CName>PyClass</CName>
    <Description>A textual description.</Description>
  </Class>
</Package>
```

5.4 Limitations

The biggest limitation if of course that there is not written support for very many modules. The Tkinter module is limited to the classes listen in section 5.2.5, and not all built in methods have been implemented in the `builtin` namespace either. The supported functions are listed in the configuration file in appendix B.

A limitation of the qGen program itself is that exception handling is not supported. This is something that can be avoided in most cases and has not

been a priority in this project. There is nothing in the program that prevents exception handling, so support for it can be added at a later stage.

Lists and dictionaries can not contain user-defined instances of classes. The `PyList` and `PyDict` types support the use of any type of element, but to be able to extract the class, methods that handle type-casting of the class would have to be spawned. This is something that is not yet implemented.

The program currently only supports translating programs in one source file. The program treats all imported modules as "external" and will look for them in the library. It will not load the source code of imported packages and attempt to translate these automatically. Adding this feature is the main thing that should be done when further developing this program. If the program imports a module where the source code is available, the application should call itself with that source code as argument. The target code should be added to a temporary library and be available for inclusion when further processing the program that imported the module.

Chapter 6

Conclusion

6.1 Final thoughts

While working on this project I feel that I have gained a lot of programming experience. Python and C++ are two very different languages and different techniques needs to be used when coding programs. I had had very little experience with these two languages before I started this project; I am more of a Java kind of guy and I mostly use Perl for scripting purposes. I think it is very helpful for a programmer to be proficient in more than one language, so I feel that this project has been very useful for me personally.

6.2 Similar projects

It is natural to compare this project with a few other projects. Greg Stein and Bill Tutt ran a project called py2c a while ago. It translated Python programs into C. This site where this project used to reside is removed , but I read the project while it was still up about a year or so ago (`http://lima.mudlib.org/~rassilon/py2c/` [7]). The program translated any legal Python code to C. This program outputs very verbose C code and makes use of the Python/C API which is something I did not want to do in this project.

On `sourceforge.net` [14] there are two other projects of interest. One is py2cmod which is designed "to aid in the conversion of Python modules to C extension modules while keeping the same interface into the module". This also uses the Python library and is not of much interest to me either (other than the fact that I know that other people are working on similar projects). Another project is py2cpp which aims "to generate C++ source code from Python scripts and build a standalone executable program.". This project was registered a year ago, but no files has so far been released (I have tried contacting the developer, but he has not replied).

The article "converting Python Virtual Machine Code to C" by John Aycock [1] also gives insight to the importance of having good tools to optimize Python applications by converting parts of it to C.

These projects show, if nothing else, that there are interest in the programming community for such translator programs.

6.3 Future work

I mentioned earlier that I would like to continue this work after this project is done and "sentenced". Support for more Python modules can be added simply by writing C++ implementations of them and adding them to the configuration file as described in section 5.3. I feel that this framework makes it easy to expand the program to support more and more programs. Further development of this program will of course consist of improving and expanding the main compiler program, but for the most part, the work will mainly be to write support for more Python modules.

6.4 Summary

I set out to create a framework for a Python to Qt/C++ compiler that is general enough to handle most "normal" aspects of Python programs as well as making it easy to expand the program with support for more modules. I feel that this has been achieved.

Having said that, I have not implemented support for as many modules as I had hoped. I would especially have liked to implement more of the Tkinter module to be able to test the program on larger applications with more GUI components. However, I felt it was more important to work on the framework so that adding these features later would be easier and not conflict with other modules.

The applications created are very portable as long as the C++ implementations of the Python modules contain portable code. Considering this, using the Qt toolkit for graphics has been a good idea since the programs created can be compiled for a number of platforms including the Qtopia platform running on Sharp Zaurus SL-5500.

Bibliography

[1] John Ayock. Converting python virtual machine code to c. 1998.

[2] KDOC C++ and IDL Class Documentation Tool. `http://www.ph.unimelb.edu.au/~ssk/kde/kdoc/`.

[3] Matthias Kalle DalHeimer. *Programming with Qt - Second edition*. O'Reilly, 1988.

[4] Strongly Typed A discussion about the usage of the term. `http://c2.com/cgi/wiki?StronglyTyped`.

[5] Fredrik Lundh. *Python Standard Library*. O'Reilly, 1988.

[6] Lutz Prechelt. An empirical comparison of c, c++, java, perl, python, rexx and tcl. *IEEE Computer*, 2000.

[7] Gren Stein and Bill Tutt. A General Python to C compiler. `http://lima.mudlib.org/~rassilon/py2c/`.

[8] Epydoc: A tool for generating API documentation for Python modules. `http://epydoc.sourceforge.net/`.

[9] Mike Connelly Python vs. Perl vs. Java vs. C++ Runtimes. `http://www.flat222.org/mac/bench/`, 2002.

[10] Official Monty Python Website. `http://www.montypythondvd.com`.

[11] Official Python Website. `http://www.python.org`.

[12] Official Trolltech Website. `http://www.trolltech.com`.

[13] Sharp Zaurus Website. `http://www.myzaurus.com/`.

[14] Sourceforge: The world's largest Open Source software development website. `http://sourceforget.net`.

Appendix A

Installation

This is a quick guide to the installation of the required packages to run this program.

A.1 qGen

To install the `qGen` program, unpack and extract the files from the `qGen-1.0.0.tar.gz` file; This will create the directory `qGen`. The environment variable `QGENPATH` should be set to point to this directory and the directory `QGENPATH/bin` should be added to the `PATH` environment variable. Alternatively, make a symlink to the file `QGENPATH/bin/qGen` in a directory already in path.

The package consists of these files and directories:

File	Explanation
`qGen/bin/qGen`	A symbolic link to the `qGen.py` file described below.
`qGen/bin/qGen.py`	The main program. Type `qGen --help` for usage information.
`qGen/bin/codeGenerator.py`	The CodeGenerator class which stores all target code and information extracted from the source Python program.
`qGen/bin/tools.py`	A module with miscellaneous help tools needed by the `qGen` program.
`qGen/conf/packages.xml`	The configuration file where the supported modules and their corresponding C++ header files and defined.
`qGen/lib/c_builtin.h`	A header file with C++ implementation of common Python built in functions.
`qGen/lib/c_py2c.h`	A header file with a few methods that are needed when dealing with Python lists.
`qGen/lib/c_sys.h`	A header file with implementation of methods to deal with command line arguments.
`qGen/lib/c_math.h`	A header file that interfaces the ¡math.h¿ module of C++.

file	Explanation
qGen/lib/c_keyword.h	A header file that adds support for keywords as argument to functions.
qGen/lib/c_variable.h	A header file with the Variable type which is used when there are no way of knowing the type of a variable. Like in lists and dictionaries.
qGen/lib/c_list.h	A header file with implementation of the PyList class that emulates the behavior of Python lists.
qGen/lib/c_dict.h	A header file with implementation of the PyDict class that emulates the behavior of Python dictionaries.
qGen/lib/c_tkinter.h	A header file which enables support for the Tkinter module of Python. In contains a namespace with a number of classes that emulates the behavior of Tkinter by using Qt.
qGen/lib/tkint_moc.h	The moc file of the `c_tkinter.h` file. This is created by `moc` and needed by Qt.
qGen/testprograms/ inheritance.py	The testprogram of section 4.3.1 that demonstrates the translation of classes with inheritance.
qGen/testprograms/ trapezoidrule.py	The testprogram of section 4.3.2 that demonstrates the translation of a mathematical program using the Tkinter GUI toolkit.
qGen/testprograms/ speedtest_trapezoidrule.py	The testprogram of section 4.3.2 that were used to perform the comparison of the effectiveness of the target C++ program versus the source Python program.
qGen/doc/	The directory where this document can be found in various formats.

A.2 Qt

Most recent Linux distributions contain a recent version of Qt/X11. If the package is installed correctly the environment variable QTDIR should also be also set. If not, this should be set to point to the path where Qt is installed. If the package is not already present in the system, the qt-x11-free-3.3.3.tar.gz should be downloaded from the Trolltech web page [12]. To install perform the following.

Add these environment variables to your `.profile` file (For bash, ksh, zsh and sh):

```
QTDIR=/usr/local/qt
PATH=$QTDIR/bin:$PATH
MANPATH=$QTDIR/doc/man:$MANPATH
LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
```

Then login again or re-source the `.profile` file. To install the package, do the following.

```
> tar xzf qt-x11-free-3.3.3.tar.gz
> mv qt-x11-free-3.3.3 /usr/local/qt
> cd /usr/local/qt
> ./configure
> make
```

After this (or if Qt already was installed) the program can be used to translate or compile Python programs to Qt/C++ for X11. To be able to develop for the Qtopia platform, either for the emulator that emulates the Qtopia system or to cross compile for the Sharp Zaurus, the Qtopia package should be installed. To install this from source is a complicated process as explained in the product chapter in section 4. If you are running RedHat 9 or newer, or Fedora, you can use the `rpm` package `qtopia-free-1.7.0-2rh9.i386.rpm`. This can be installed by simply running this as root.

```
> rpm -i qtopia-free-1.7.0-2rh9.i386.rpm
```

To use the libraries, a few environment variables need to be set. This depends on whether the libraries for the Qtopia emulator should be used or the ones for the Sharp cross compiler. This is well documented in README files provided with the package. However, no setup is required to use this package with the `qGen` program of this project. The configuration files generated by the program make sure that all environment variables are set. `qGen` assumes this is installed to `/opt/Qtopia` which is also the default directory of the `rpm` package. If it is installed elsewhere, it should be specified by commandline argument to `qGen`.

Appendix B

Configuration

```
<Packages>
  <Package pname='' cname='builtin' file='c_builtin.h'>
    <Method>
      <PName>range</PName>
      <CName>range</CName>
      <Type>PyList</Type>
      <Description>This is the Pyton-method for creating sequences.</Description>
    </Method>
    <Method>
      <PName>str</PName>
      <CName>str</CName>
      <Type>types.StringType</Type>
      <Description></Description>
    </Method>
    <Method>
      <PName>float</PName>
      <CName>c_float</CName>
      <Type>types.FloatType</Type>
      <Description></Description>
    </Method>
    <Method>
      <PName>int</PName>
      <CName>int</CName>
      <Type>types.IntType</Type>
      <Description></Description>
    </Method>
  </Package>

  <Package pname='math' cname='math' file='c_math.h'>
    <Method>
      <PName>sin</PName>
      <CName>sin</CName>
      <Type>types.FloatType</Type>
      <Description></Description>
    </Method>
    <Method>
      <PName>cos</PName>
      <CName>cos</CName>
      <Type>types.FloatType</Type>
      <Description></Description>
    </Method>
    <Method>
      <PName>tan</PName>
      <CName>tan</CName>
      <Type>types.FloatType</Type>
      <Description></Description>
    </Method>
  </Package>

  <Package pname='Tkinter' cname='Tkinter' file='c_tkinter.h'>
    <Class>
```

```
      <PName>Tk</PName>
      <CName>Tk</CName>
      <Description></Description>
    </Class>
    <Class>
      <PName>Frame</PName>
      <CName>Frame</CName>
      <Description></Description>
    </Class>
    <Class>
      <PName>StringVar</PName>
      <CName>StringVar</CName>
      <Description></Description>
    </Class>
    <Class>
      <PName>Entry</PName>
      <CName>Entry</CName>
      <Description></Description>
    </Class>
    <Class>
      <PName>Button</PName>
      <CName>Button</CName>
      <Description></Description>
    </Class>
    <Class>
      <PName>Label</PName>
      <CName>Label</CName>
      <Description></Description>
    </Class>
  </Package>

  <Package pname='list' cname='list' file='c_list.h'>
  </Package>

  <Package pname='variable' cname='variable' file='c_variable.h'>
  </Package>

  <Package pname='sys' cname='sys' file='c_sys.h'>
  </Package>

</Packages>
```

Appendix C

Test programs

C.1 Inheritance

C.1.1 inheritance.py

```
#!/usr/bin/env python

class Super:
    def setName(self, newname):
        self.name = str(newname)

    def getName(self):
        return self.name

class A(Super):
    def __init__(self):
        self.name = "First"

class B(Super):
    def __init__(self):
        self.name = "Second"

class C(Super):
    def __init__(self):
        self.name = "Third"

a = A()
a.setName("1st")
b = B()
b.setName("2nd")
c = C()
c.setName("3rd")

x = a.getName()
y = b.getName()
z = c.getName()

print x, y, z
```

inheritance.py.[ABC].h

```
 /* Includes */
#include <stdio.h>
#include <iostream>
#include <string>
#include "c_builtin.h"



/* Class definitions */
class A : public Super {
public:
  A();

  string name;
}; // end of class A

/* Functions of class A */
A::A() {
  A *self = this;
    self->name = "First";
} // end of function A




/* Class definitions */
class B : public Super {
public:
  B();

  string name;
}; // end of class B

/* Functions of class B */
B::B() {
  B *self = this;
    self->name = "Second";
} // end of function B




/* Class definitions */
class C : public Super {
public:
  C();

  string name;
}; // end of class C

/* Functions of class C */
C::C() {
  C *self = this;
    self->name = "Third";
} // end of function C
```

inheritance.py.Super.h

```
 /* Includes */
#include <stdio.h>
#include <iostream>
#include <string>
#include "c_builtin.h"

/* Class definitions */
class Super {
public:
  string getName();
  Super();
  void setName(string newname);

  string name;
}; // end of class Super

/* Functions of class Super */
string Super::getName() {
  Super *self = this;
  return self->name;
} // end of function getName

Super::Super() {
} // end of function Super

void Super::setName(string newname) {
  Super *self = this;
  self->name = builtin::str(newname);
} // end of function setName
```

inheritance.py.main.cpp

```
 /* Includes */
#include <stdio.h>
#include <iostream>
#include <string>
#include "c_builtin.h"
#include "inheritance.py.Super.h"
#include "inheritance.py.A.h"
#include "inheritance.py.C.h"
#include "inheritance.py.B.h"

/* Global variables declarations */
A a; C c; B b;
string y; string x; string z;

/* All functions */
int main(int argc, char* argv[]) {
  a = A();
  a.setName("1st");
  b = B();
  b.setName("2nd");
  c = C();
  c.setName("3rd");
  x = a.getName();
  y = b.getName();
  z = c.getName();
  std::cout << x << " " << y << " " << z << std::endl;
  return 0;
} // end of function main
```

C.2 Trapezoid Rule

C.2.1 trapezoidrule.py

```
 #!/ifi/ganglot/k00/in228/www_docs/packages/Linux/bin/python
import math, Tkinter

class Functions:
    def f(self, x):
        """This is the function sin(x)/x. If x == 0, then it evaluates to 1."""
        s = math.sin(x)
        if x == 0:  return 1
        ret = s/x ; return ret
    def g(self, x):
        """This is the function x*x"""
        a = x * x ; return a
    def h(self, x):
        """This is the function 3^x"""
        a = math.pow(3,x); return a

def trapezoid(a, b, n):
    """Definite integral of f(x) for x in [a, b]."""
    """Calculating the intervals."""
    h = (b - a) / float(n-1)
    """Figuring out which function to use og making the calculation."""
    func = Functions()
    if int(activeFunc.get()) == 2:
        sum = 0.5 * (func.f(a) + func.h(b))
        for i in range(1,n-1):
            x = a + int(i) * h
            sum = sum + float(func.f(x))
    elif int(activeFunc.get()) == 1:
        sum = 0.5 * (func.f(a) + func.f(b))
        for i in range(1,n-1):
            x = a + int(i) * h
            sum = sum + float(func.g(x))
    elif int(activeFunc.get()) == 0:
        sum = 0.5 * (func.f(a) + func.f(b))
        for i in range(1,n-1):
            x = a + int(i) * h
            sum = sum + float(func.f(x))
    """Multiplying with the interval size."""
    sum = sum * h
    return sum

def execute():
    result.set("%g" % trapezoid(int(stv_a.get()), int(stv_b.get()), int(stv_n.get())))
tk = Tkinter.Tk() ; frame = Tkinter.Frame(tk) ; frame.grid(row=0, column=0)
activeFunc = Tkinter.StringVar() ; activeFunc.set(0)
t_label = Tkinter.Label(frame, text="Trapezoid rule", width=15, relief="sunken")
t_label.grid(row=0, column=0)
func_f = Tkinter.Radiobutton(frame, text="f(x) = (sin x) / x", variable=activeFunc, value="0")
func_g = Tkinter.Radiobutton(frame, text="g(x) = x^2", variable=activeFunc, value="1")
func_h = Tkinter.Radiobutton(frame, text="h(x) = 3^x", variable=activeFunc, value="2")
func_f.grid(row=1, column=0) ; func_g.grid(row=2, column=0) ; func_h.grid(row=3, column=0)
stv_n = Tkinter.StringVar() ; stv_a = Tkinter.StringVar() ; stv_b = Tkinter.StringVar()
stv_n.set(1024) ; stv_a.set(0) ; stv_b.set(1)
frame_n = Tkinter.Frame(frame) ; frame_a = Tkinter.Frame(frame) ; frame_b = Tkinter.Frame(frame)
frame_n.grid(row=4, column=0) ; frame_a.grid(row=5, column=0) ; frame_b.grid(row=6, column=0)
lab_n = Tkinter.Label(frame_n, text="n") ; lab_a = Tkinter.Label(frame_a, text="a")
lab_b = Tkinter.Label(frame_b, text="b")
ent_n = Tkinter.Entry(frame_n, textvariable=stv_n, width=12, relief="sunken")
ent_a = Tkinter.Entry(frame_a, textvariable=stv_a, width=12, relief="sunken")
ent_b = Tkinter.Entry(frame_b, textvariable=stv_b, width=12, relief="sunken")
lab_n.grid(row=0, column=0) ; lab_a.grid(row=0, column=0) ; lab_b.grid(row=0, column=0)
ent_n.grid(row=0, column=1) ; ent_a.grid(row=0, column=1) ; ent_b.grid(row=0, column=1)

btnExecute = Tkinter.Button(frame, text="Calculate", command=execute)
btnExecute.grid(row=7, column=0) ; result = Tkinter.StringVar()
labResult = Tkinter.Label(frame, textvariable=result, width=30, relief="sunken")
labResult.grid(row=8, column=0)
tk.mainloop()
```

trapezoidrule.py.Function.h

```
/**
  * Code generated by qGen 1.0 by Martin Jensen
  *
  * File: trapezoidrule/trapezoidrule.py.Functions.h
  *
  **/

/* Includes */
#include <stdio.h>
#include <iostream>
#include <string>
#include "c_builtin.h"
#include "c_math.h"
#include "c_tkinter.h"

/* Class definitions */
class Functions {
public:
  double h(double x);
  Functions();
  double g(double x);
  double f(double x);

}; // end of class Functions


/* Functions of class Functions */
double Functions::h(double x) {
  double a;
  Functions *self = this;
  a = math::pow(3, x);
  return a;
} // end of function h

Functions::Functions() {
} // end of function Functions

double Functions::g(double x) {
  double a;
  Functions *self = this;
  a = (x * x);
  return a;
} // end of function g

double Functions::f(double x) {
  double s;
  double ret;
  Functions *self = this;
  s = math::sin(x);
  if ( x == 0 ) {
    return 1;
  } // end of if
  ret = (s / x);
  return ret;
} // end of function f
```

trapezoidrule.py.main.cpp [1]

```
 /* Includes */
#include <stdio.h>
#include <iostream>
#include <string>
#include "c_builtin.h"
#include "c_math.h"
#include "c_tkinter.h"
#include "trapezoidrule.py.Functions.h"

/* Global variables declarations */
Tkinter::Tk tk;
Tkinter::Frame frame;
Tkinter::Frame frame_a; Tkinter::Frame frame_b; Tkinter::Frame frame_n;
Tkinter::StringVar stv_a; Tkinter::StringVar stv_b; Tkinter::StringVar stv_n;
Tkinter::StringVar activeFunc; Tkinter::StringVar result;
Tkinter::Label lab_n; Tkinter::Label lab_a; Tkinter::Label lab_b;
Tkinter::Label labResult; Tkinter::Label t_label;
Tkinter::Entry ent_n; Tkinter::Entry ent_b; Tkinter::Entry ent_a;
Tkinter::Radiobutton func_h; Tkinter::Radiobutton func_g; Tkinter::Radiobutton func_f;
Tkinter::Button btnExecute;

/* Abstract function declarations */
double trapezoid(double a, double b, double n);
void execute();

/* All functions */
double trapezoid(double a, double b, double n) {
  double h; double sum; Functions func; double x;
  /* "Calculating the intervals." */
  h = ((b - a) / builtin::c_float((n - 1)));
  /* "Figuring out which function to use og making the calculation." */
  func = Functions();
  if ( builtin::c_int(activeFunc.get()) == 2 ) {
    sum = (0.5 * (func.f(a) + func.h(b)));
    PyList list_aGsRt = builtin::range(1, (n - 1));
    for (Variable i = list_aGsRt.initialize(); list_aGsRt.hasMore();
         i = list_aGsRt.nextElement()) {
      x = (a + (builtin::c_int(i) * h));
      sum = (sum + builtin::c_float(func.f(x)));
    } // end of for
  } else if ( builtin::c_int(activeFunc.get()) == 1 ) {
    sum = (0.5 * (func.f(a) + func.f(b)));
    PyList list_eqBwZ = builtin::range(1, (n - 1));
    for (Variable i = list_eqBwZ.initialize(); list_eqBwZ.hasMore();
         i = list_eqBwZ.nextElement()) {
      x = (a + (builtin::c_int(i) * h));
      sum = (sum + builtin::c_float(func.g(x)));
    } // end of for
  } else if ( builtin::c_int(activeFunc.get()) == 0 ) {
    sum = (0.5 * (func.f(a) + func.f(b)));
    PyList list_mGsOp = builtin::range(1, (n - 1));
    for (Variable i = list_mGsOp.initialize(); list_mGsOp.hasMore();
         i = list_mGsOp.nextElement()) {
      x = (a + (builtin::c_int(i) * h));
      sum = (sum + builtin::c_float(func.f(x)));
    } // end of for
  } // end of if
  /* "Multiplying with the interval size." */
  sum = (sum * h);
  return sum;
} // end of function trapezoid

void execute() {
  char buf_pFPaFHWz[50];
  string pFPaFHWz;
  sprintf(buf_pFPaFHWz, "%g", trapezoid(builtin::c_int(stv_a.get()),
          builtin::c_int(stv_b.get()), builtin::c_int(stv_n.get())));
  pFPaFHWz = buf_pFPaFHWz;
  result.set(pFPaFHWz);
} // end of function execute
```

trapezoidrule.py.main.cpp [2]

```
int main(int argc, char* argv[]) {
  tk.init(argc, argv);
  frame.init(tk);
  Keyword key_TfBSBaDx;
  key_TfBSBaDx.add("column", 0);
  key_TfBSBaDx.add("row", 0);
  frame.grid(key_TfBSBaDx);
  activeFunc.init();
  activeFunc.set(0);
  Keyword key_LUXiOoNa;
  key_LUXiOoNa.add("text", "Trapezoid rule");
  key_LUXiOoNa.add("relief", "sunken");
  key_LUXiOoNa.add("width", 15);
  t_label.init(frame, key_LUXiOoNa);
  Keyword key_STtvmmCG;
  key_STtvmmCG.add("column", 0);
  key_STtvmmCG.add("row", 0);
  t_label.grid(key_STtvmmCG);
  Keyword key_wLRKzAPP;
  key_wLRKzAPP.add("variable", &activeFunc);
  key_wLRKzAPP.add("text", "f(x) = (sin x) / x");
  key_wLRKzAPP.add("value", "0");
  func_f.init(frame, key_wLRKzAPP);
  Keyword key_dkqBnUOO;
  key_dkqBnUOO.add("variable", &activeFunc);
  key_dkqBnUOO.add("text", "g(x) = x^2");
  key_dkqBnUOO.add("value", "1");
  func_g.init(frame, key_dkqBnUOO);
  Keyword key_xGEGEhZc;
  key_xGEGEhZc.add("variable", &activeFunc);
  key_xGEGEhZc.add("text", "h(x) = 3^x");
  key_xGEGEhZc.add("value", "2");
  func_h.init(frame, key_xGEGEhZc);
  Keyword key_UJSiUcdw;
  key_UJSiUcdw.add("column", 0);
  key_UJSiUcdw.add("row", 1);
  func_f.grid(key_UJSiUcdw);
  Keyword key_PUFTdLwL;
  key_PUFTdLwL.add("column", 0);
  key_PUFTdLwL.add("row", 2);
  func_g.grid(key_PUFTdLwL);
  Keyword key_uQzsVOfG;
  key_uQzsVOfG.add("column", 0);
  key_uQzsVOfG.add("row", 3);
  func_h.grid(key_uQzsVOfG);
  stv_n.init();
  stv_n.set(1024);
  stv_a.init();
  stv_a.set(0);
  stv_b.init();
  stv_b.set(1);
  frame_n.init(frame);
  frame_a.init(frame);
  frame_b.init(frame);
  Keyword key_kVRaKDZp;
  key_kVRaKDZp.add("column", 0);
  key_kVRaKDZp.add("row", 4);
  frame_n.grid(key_kVRaKDZp);
  Keyword key_ybkiUiMF;
  key_ybkiUiMF.add("column", 0);
  key_ybkiUiMF.add("row", 5);
  frame_a.grid(key_ybkiUiMF);
  Keyword key_FUvXxDTs;
  key_FUvXxDTs.add("column", 0);
  key_FUvXxDTs.add("row", 6);
```

trapezoidrule.py.main.cpp [3]

```
  frame_b.grid(key_FUvXxDTs);
  Keyword key_jLSPJJmv;
  key_jLSPJJmv.add("text", "n");
  lab_n.init(frame_n, key_jLSPJJmv);
  Keyword key_uELCGWzm;
  key_uELCGWzm.add("text", "a");
  lab_a.init(frame_a, key_uELCGWzm);
  Keyword key_lRntYfkb;
  key_lRntYfkb.add("text", "b");
  lab_b.init(frame_b, key_lRntYfkb);
  Keyword key_svoksvEa;
  key_svoksvEa.add("width", 12);
  key_svoksvEa.add("textvariable", &stv_n);
  key_svoksvEa.add("relief", "sunken");
  ent_n.init(frame_n, key_svoksvEa);
  Keyword key_RYfyijtq;
  key_RYfyijtq.add("width", 12);
  key_RYfyijtq.add("textvariable", &stv_a);
  key_RYfyijtq.add("relief", "sunken");
  ent_a.init(frame_a, key_RYfyijtq);
  Keyword key_jswWHxRM;
  key_jswWHxRM.add("width", 12);
  key_jswWHxRM.add("textvariable", &stv_b);
  key_jswWHxRM.add("relief", "sunken");
  ent_b.init(frame_b, key_jswWHxRM);
  Keyword key_fbtzefAP;  key_fbtzefAP.add("column", 0);  key_fbtzefAP.add("row", 0);
  lab_n.grid(key_fbtzefAP);
  Keyword key_gPRwpXQx;
  key_gPRwpXQx.add("column", 0);
  key_gPRwpXQx.add("row", 0);
  lab_a.grid(key_gPRwpXQx);
  Keyword key_zEGzCvsI;
  key_zEGzCvsI.add("column", 0);
  key_zEGzCvsI.add("row", 0);
  lab_b.grid(key_zEGzCvsI);
  Keyword key_sXjzzECE;
  key_sXjzzECE.add("column", 1);
  key_sXjzzECE.add("row", 0);
  ent_n.grid(key_sXjzzECE);
  Keyword key_BODuWdnZ;
  key_BODuWdnZ.add("column", 1);
  key_BODuWdnZ.add("row", 0);
  ent_a.grid(key_BODuWdnZ);
  Keyword key_mPVsKyaO;  key_mPVsKyaO.add("column", 1);  key_mPVsKyaO.add("row", 0);
  ent_b.grid(key_mPVsKyaO);
  Keyword key_VziqwNjo;
  btnExecute.addCommand(execute);
  key_VziqwNjo.add("text", "Calculate");
  btnExecute.init(frame, key_VziqwNjo);
  Keyword key_HStxSeDO;
  key_HStxSeDO.add("column", 0);
  key_HStxSeDO.add("row", 7);
  btnExecute.grid(key_HStxSeDO);
  result.init();
  Keyword key_bafDKbKo;
  key_bafDKbKo.add("width", 30);
  key_bafDKbKo.add("textvariable", &result);
  key_bafDKbKo.add("relief", "sunken");
  labResult.init(frame, key_bafDKbKo);
  Keyword key_MARaMgoA;
  key_MARaMgoA.add("column", 0);
  key_MARaMgoA.add("row", 8);
  labResult.grid(key_MARaMgoA);
  tk.mainloop();
  return 0;
} // end of function main
```

Appendix D

PyDoc

D.1 Module qGen

```
A Python to Qt/C++ compiler

qGen is used for translating Python programs into Qt/C++.

Usage:
  qGen <input-file> [options]

e.g.
  qGen examples/example1.class.py -o output/example1
```

Functions

arithmeticExpr(*node*)

Returns a string representing the expression in the node.
This can be any valuetype. Expressions and functions.
To make sure everything goes right, I'm priting out a full set of parenthesis for arithmetic expressions.

compareExpr(*node*)

Handles comparisons. It returns the textual representation of the comparison.

debug(*message*)

Writes a debug message in the target code.
This is only used for debugging during development

determineType(*node*)

Determines type type of the given node and returns it.

functionStatement(*node*)

Determines the type of node and calls the appripriate function.
This function is called by the nodeStmt function for each node in the Stmt node. These statements are only leagal within functions.

getDeclaration(*variableName*)

Checks whether a variable has been declared and returns the declatation type if it exists.

getReturnType(*function*)

Tries to find out what the returntype of a function is, and returns it if possible.

isClass(*function*)

Determines wheter the functionCall actually is a call to a class

keywordVar()

Handles a collection of keywords that has been built by the function nodeKeyword.
This method is called after all keywords in an argumentslist has been processed. Then all then keywords are added to an instance of the Keyword class. This instance is then returned by this function.

loopStatement(*node*)

Determines the type of node and calls the appripriate function.
This function is called by the nodeStmt function for each node in the Stmt node. These statements are only leagal within a loop.

makeFunctionDeclaration(*functionName*)

Makes a declaration of a function.
First it checks if it's declared before. If it is, then it does nothing.

nodeAssign(*node*)

Handles assignments statements and writes to the codegenerator.
Determines the type of value to be assigned and declares the variable if needed. If the variable exists and the assignment is of a different type, it is replaced by a new variable.

nodeAugAssign(*node*)

Handles augmented assignments. This writes target code to the codegenerator.

nodeBreak(*node*)

Handles break statements. This writes code to the codegenerator.
To end a loop. This is all there is to it. Doesn't even have any children.

nodeCallFunc(*node*)

Handles the callfunc-statement. Return a string-representation of the call. This check if the function is defined in the source code or if it is a pre-made function. It looks the function up in a table from the tools module and gets the return type.

nodeClass(*node*)

Handles creation of classes. This writes code to the codegenerator.
Defines a class with inheritance. It calls upon the function nodeStmt to evaluate the statements of the class.

nodeConst(*node*)

Handles a constants like strings or numbers.
This simply returns the content of the constant. If the constant is a string, then it is quoted.

nodeContinue(*node*)

Handles continue statements. This writes code to the codegenerator.
To start at the next loop-cycle without finishing the current one. This is all there is to it. Doesn't even have any children.

nodeDict(*node*)

Handles the creation of a dictionary.
If the dictionary contains initialization values, then these are added to a temporary dictionary. This dictionary is then returned by this function. This way the dictionary can be used as arguments and in any arithmetic expression.

nodeDiscard(*node*)

Handles the discard-statement. This writes code to the codegenerator.
This is called when a function or value is evaluated, but the return value should not be used.

nodeFor(*node*)

Handles for-statements. This writes code to the codegenerator.
This works a bit differently in C++ than Python. The compiler will try to convert whatever it is iterating over to a PyList and create an iterator that that traverses through the list.

nodeFunction(*node*)

Handles creation of functions. This writes code to the codegenerator.
This evaluates all arguments given and creates the header of the function with argument list and name. The return type is determined if a return-statement is encountered when handling the statements of the function. If no return statement is encounteres, the function is assumed to be a void-function. The types of the arguments are also expected to become clear as the statements of the function are evaluated.

nodeGetattr(*node*)

Handles the discard-statement. Return a string representation of the node. Used when calling a function or a variable within a class or module. This can be nested so the method is recursive.

nodeIf(*node*)

Handles if statements. This writes code to the codegenerator.
This evaluates each part of the if statement and calls upon the nodeStmt()-method for handling the statements.

nodeImport(*node*)

Handles import statements. This writes code to the codegenerator.
When a package is imported, the corresponding c-package is imported in the target C++-code. The C++-packages contains all the same methods as the Python package. The packages and methods and the corresponding C++-packages and methods are configured in the configuration-file.

nodeKeyword(*node*)

Handles keywords that is used as arguments to a function.
When a keyword is encountered, it is stored in a keystore. These keywords are used after all the keywords in the argumentlist are processed. They are extracted in the keywordVar-function.

nodeList(*node*)

Handles the creation of a list.
If the list contains initialization values, then these are added to a temporary list. This temporary list is then returned by this function. This way the list can be used as arguments and in any arithmetic expression.

nodeName(*node*)

Handles a referance to a variable. This returns a string representing the variable.
If the variable is of a 'normal' type, then the name is simply returned. If it is a generic variable it have an appendix appended to make sure that the value is properly extracted.

nodePass(*node*)

Handles pass statements. This writes code to the codegenerator.
This statement has no meaning in C++ that uses {} for block structure. In the target code, this is simply representet by a comment.

nodePrintnl(*node*)

Handles print statements. This writes code to the codegenerator.
The arguments to the statement are written to the standard output.
The children of the node is evaluated by the arithemticExpr()-method.

nodeReturn(*node*)

Handles return statements. This writes code to the codegenerator.
In addition to writing the actual return statement to the codegenerator, this method also tells the method what return type it is supposed to be defined with.

nodeStmt(*node*)

Handles a collection of statements.
This is called from within loops, functions, classes, etc. It can contain an arbitrary number of nodes, each representing a statement. The functions calls upon the statement-methods for each of the nodes.

nodeSubscript(*node*)

Handles a subscript operation on a list or a dictionary.
This is not directly translatable without overloading the []-operator. Instead it uses methods like getString and getInt together with the number from the subscript operation to perform the action. Like this:
print list[4] => std::cout << list.getString(4) << std::endl; int(dict['two']) => list.getInt('two');

nodeUnarySub(*node*)

Handles negative constants.
All this does is to place a negative (-) sign in front of a constant.

nodeWhile(*node*)

Handles while-statements. This writes code to the codegenerator.
This evaluates each part of the while statement and calls upon the nodeStmt()- method for handling the statements.

statement(*node*)

Determines the type of node and calls the appripriate function.
This function is called by the nodeStmt function for each node in the Stmt node as well as by the main loop of the program. These statements are leagal globally.

Variables

Name	Description
__author__	**Value:** `'Martin Jensen'` *(type=str)*
__version__	**Value:** `'1.0'` *(type=str)*

D.2 Module codeGenerator

support module for qGen.

This class contains variables and methods to store the classes, functions, variables and all target code produced by qGen.

Variables

Name	Description
__author__	**Value:** `'Martin Jensen'` *(type=str)*
__version__	**Value:** `'1.0'` *(type=str)*

Class CodeGenerator

This is the classgenerator

This class contains variables and methods to store the classes, functions, variables and all target code produced by qGen.

Methods

__init__(*self*)

Initializes the CodeGenerator class
This initialyzes quite a few lists and dictionaries used when translating the program.

addClass(*self*, *className*, *inherits*)

Adds a class to the target code.
className is the name of the class. inherits is a list of the classes this class should inherit from.

addFunction(*self*, *functionName*, *fClass*, *arguments*, *argOrder*, *initValues*, *initTypes*)

Adds a function to the target code. If a class is specfied, then it is added to that class. If not, it adds the function to the current class. If there are no current class, then the functions is outside any class.
functionsName is the name of the function. fClass is the class the functions should belong to (if any). arguments are the arguments of the function. argOrder is the order the arguments should come in. initValues are the initialization values of the arguments. initTypes are the types of the initialization values.

addImport(*self*, *imp*)

Adds the specified package to the list of packages to be imported.
imp is the package to import.

addLine(*self*, *line*)

Adds a line to the target code.
This is called by the qGen program whenever it has a code to add to the program. The functions checks where in the program the line should be added and on what block-level and writes it to the appropriate class and function.

addReturnType(*self*, *type*)

Sets the return type of the current function.

appendCode(*self*, *line*)

Takes care of code that should be appended after the next line of code. The code that is sent here written _after_ the next normal line of code is written.

changeVarType(*self*, *varName*, *varType*)

Used to alter the type of a variable. If a variable changes type, then a new variable is spawned within the same scope. The old variable is replaced by the new from this point on.

curFunc(*self*)

Returns the function the qGen program is currently processing.

declareArgument(*self*, *val*, *valType*=<type 'NoneType'>, *clss*=None)

This sets the type of an argument to the current function.
val is the variable's name. valType is the type of the variable. clss is the class.

decreaseBlocklevel(*self*)

Decreases the blocklevel

endClass(*self*)

Closes the current class.
This updates the current class name.

endFunction(*self*)

Closes the current function.
This decreases the blocklevel, and removes the name of the function as the current function name.

getCode(*self*)

Returns all the code generated so far.
This is only used during development.

getCurrentVarName(*self*, *varName*)

Finds out if the variable has changed type. If it has, it will return the name of the new variable. If not it returns the original name.

getDeclaration(*self*, *varName*, *clss*=`None`)

Used to determine the type of the variable if it's declared. This returns None if the specified variable is undeclared, or NoneType if the variable is used in the argumentlist.

getReturnType(*self*, *funcName*, *className*=`None`)

Returns the return type of the given function.
funcName is the name of the function. className is the name of the class the function is in (if any).

getTheType(*self*)

Gets the type of the last used variable of the program.

increaseBlocklevel(*self*)

Increases the blocklevel

initialization(*self*)

This handles additional initializations that is required.

isPointer(*self*, *varName*)

Returns true if the variable should be treated as a pointer. Returns false if it should be treated as a referance or an object.

makeDeclaration(*self*, *varName*, *varType*, *clss*=`None`)

Declare a variable.
This checks where the variable should be declared and adds it to that class or function. This also checks if the variable is an argument to a function in which case no declaretion is made, only the type of the arguement is updated.

setCurrentVariable(*self*, *currentVar*)

Saves the name of the variable the qGen program is currently working on.

setTheType(*self*, *type*)

Saves the type of the last used variable of the program.

undeclare(*self*, *varName*)

Used to undeclare a variable. This is only used in loops where a variable should only be valid within the loop

writeCode(*self*, *outputdir*, *prefix*='')

Writes all the target code to appropriate files.
This method returns a list of the files created.

Class Variables

Name	Description

D.3 Module tools

help methods that are used by the qGen program.

This class consists of help functions and configuration functions for the qGen program.

Variables

Name	Description
__author__	**Value:** `'Martin Jensen'` *(type=`str`)*
__version__	**Value:** `'1.0'` *(type=`str`)*

Class Tools

Tools class that contains help methods that are used by the qGen program.

This class consists of help functions and configuration functions for the qGen program.

Methods

__init__(*self*)

Initializing the list of functions.

addPackage(*self*, *packageName*)

When a package is imported, the content is added to the list of visible functions.

castError(*self*, *message*, *fatal*=0)

Adds the error message to the list of error messages.
The program will attempt to continue processing the file unless the argument fatal is given.

convertMetType(*self*, *type*)

Returns the textual representation of the type of a function.

convertType(*self*, *type*)

Returns the textual representation of the type of a variable.

getErrormessages(*self*)

Returns the error messages that has been encountered during the processing of the file.

readConfig(*self*)

Reads the configuration from an xml-file
This method reads the package-configuration from the config-xml-file. The packages is used to find the correct methods in the C++-library to replace with the calls to Python-methods.

requireArgument(*self*, *name*)

Adds required arguments to the function.
Some functions require specific argument to be given. This functions makes sure they do in those cases. This is called for the callFunc method in the qGen.py file.

usage(*self*)

Returns the usage of the program.

Appendix E

C++Doc

E.1 Class List

E.2 Class Hierachy

Tkinter
QObject (unknown)
Tk_Object
Button
Entry
Frame
Label
Radiobutton
StringVar
Tk
TextObject
Entry
Label
Variable
PyDict
PyList
PySys
Keyword
builtin
math
py2c

E.3 namespace Tkinter

Adds support for the Tkinter module to the qGen program.

Public Types

- class ***TextObject*** 78

Public Methods

Detailed Description

This namespace contains all supported classes in the Tkinter module. These classes correspond to classes in the Python Tkinter module. Each of the widget-classes present here is mapped to a widget in Qt.

Version:

1.0

Author:

Martin Jensen

Member Documentation

Tkinter::TextObject (class)

The super class of the widgets that can have a string variable assigned to them.

Tkinter::Tk_Object (class)

The super class of all widgets in this namespace. This defines common methods and variables. It inherits QObject for Qt-support.

Tkinter::Tk (class)

The main class in the hierarchy of widgets. QApplication is attached to this class.

Tkinter::Frame (class)

A frame which other widget can be assigned to. QFrame is attached to this class.

Tkinter::StringVar (class)

A string objects that can be assigned other widgets. QString is attached to this class.

Tkinter::Button (class)

A button that can be assigned to a function. QPushButton is attached to this class.

Tkinter::Radiobutton (class)

A radiobutton that can be assigned to a function and grouped with other radiobuttons. QRadioButton is attached to this class.

Tkinter::Label (class)

A label that can present text on the canvas. QLabel is attached to this class.

Tkinter::Entry (class)

A textfield where users can input data. QLineEdit is attached to this class.

QGridLayout* `Tkinter::Tk_Object::getGridLayout` () []

QGridLayout* `Tkinter::Tk::getGridLayout` () []

QGridLayout* `Tkinter::Frame::getGridLayout` () []

void `Tkinter::Tk_Object::adjust` () []

void `Tkinter::Tk::adjust` () []

void `Tkinter::Frame::adjust` () []

void `Tkinter::Tk_Object::setLoglevel` (**int l**) []

int `Tkinter::Tk_Object::getLoglevel` () []

void `Tkinter::Tk_Object::error` (**string s**) []

void `Tkinter::Tk_Object::warn` (**string s**) []

void `Tkinter::Tk_Object::info` (**string s**) []

void `Tkinter::Tk_Object::debug` (**string s**) []

`Tkinter::Tk::Tk` () []

`Tkinter::Button::Button` () []

`Tkinter::Radiobutton::Radiobutton` () []

`Tkinter::Frame::Frame` () []

`Tkinter::Label::Label` () []

`Tkinter::Entry::Entry` () []

`Tkinter::StringVar::StringVar` () []

`Tkinter::Tk::~Tk` () []

`Tkinter::Button::~Button` () []

`Tkinter::Radiobutton::~Radiobutton` () []

`Tkinter::Frame::~Frame` () []

`Tkinter::Label::~Label` () []

`Tkinter::Entry::~Entry` () []

`Tkinter::StringVar::~StringVar` () []

void `Tkinter::Tk::init` (**int argc, char* argv**[]) []

void `Tkinter::Button::init` (**Tk_Object &p**) []

void `Tkinter::Radiobutton::init` (**Tk_Object &p**) []

void `Tkinter::Frame::init` (**Tk_Object &p**) []

void `Tkinter::Label::init` (**Tk_Object &p**) []

void `Tkinter::Entry::init` (**Tk_Object &p**) []

void `Tkinter::StringVar::init` () []

void `Tkinter::Button::init` (**Tk_Object &p, Keyword& k**) []

void `Tkinter::Radiobutton::init` (**Tk_Object &p, Keyword& k**) []

void `Tkinter::Frame::init` (**Tk_Object &p, Keyword& k**) []

void `Tkinter::Label::init` (**Tk_Object &p, Keyword& k**) []

void `Tkinter::Entry::init` (**Tk_Object &p, Keyword& k**) []

QWidget* `Tkinter::Tk::getWidget` () []

`#include <c_tkinter.h>`
Inherits: Tk_Object

public

()

Public Methods

Protected Methods

Protected Members

Detailed Description

A button that can be assigned to a function. QPushButton is attached to this class.

Member Documentation

`Tkinter::Button::Button` **()** []

Constructor: Does nothing except logging.

`Tkinter::Button::~Button` **()** []

Destructor: Does nothing except logging.

void `Tkinter::Button::init` **(Tk_Object&)** []

Creates the QButton widget and sets the widget of the Tk_Object p as its parent.

void `Tkinter::Button::init` **(Tk_Object&, Keyword&)** []

Creates the QButton widget and sets the widget of the Tk_Object p as its parent. This also reads the keyword arguments to the method.

void `Tkinter::Button::pack` **()** []

Packs the widget using the pack layout manager. This is poorly supported. Use the grid-methods instead.

Reimplemented from *Tk_Object*

E.5 class Tkinter::Entry

A textfield where users can input data.

`#include <c_tkinter.h>`

Inherits: TextObject

private

(), Tk_Object

public

()

Public Methods

Protected Methods

Detailed Description

A textfield where users can input data. QLineEdit is attached to this class.

Member Documentation

`Tkinter::Entry::Entry` () []

Constructor: Does nothing except logging.

`Tkinter::Entry::~Entry` () []

Destructor: Does nothing except logging.

void `Tkinter::Entry::init` **(Tk_Object&)** []

Creates the QLineEdit widget and sets the widget of the Tk_Object p as its parent.

void `Tkinter::Entry::init` **(Tk_Object&, Keyword&)** []

Creates the QLineEdit widget and sets the widget of the Tk_Object p as its parent. This also reads the keyword arguments to the method.

void `Tkinter::Entry::pack` () []

Packs the widget using the pack layout manager. This is poorly supported. Use the grid-methods instead.

Reimplemented from *Tk_Object*

E.6 class Tkinter::Frame

A frame which other widget can be assigned to.

`#include <c_tkinter.h>`

Inherits: Tk_Object

public

()

Public Methods

Protected Methods

- QWidget* **getWidget** () .....**??**
- QGridLayout* **getGridLayout** () .....**??**

Detailed Description

A frame which other widget can be assigned to. QFrame is attached to this class.

Member Documentation

`Tkinter::Frame::Frame` () []

Constructor: Does nothing except logging.

`Tkinter::Frame::~Frame` () []

Destructor: Does nothing except logging.

void `Tkinter::Frame::init` **(Tk_Object&)** []

Creates the QFrame widget and sets the widget of the Tk_Object p as its parent.

void `Tkinter::Frame::init` **(Tk_Object&, Keyword&)** []

Creates the QFrame widget and sets the widget of the Tk_Object p as its parent. This also reads the keyword arguments to the method.

void `Tkinter::Frame::pack` () []

Packs the widget using the pack layout manager. This is poorly supported. Use the grid-methods instead.

Reimplemented from *Tk_Object*

E.7 class Tkinter::Label

A label that can present text on the canvas.

`#include <c_tkinter.h>`

Inherits: TextObject

private

(), Tk_Object

public

()

Public Methods

Protected Methods

Detailed Description

A label that can present text on the canvas. QLabel is attached to this class.

Member Documentation

`Tkinter::Label::Label` () []

Constructor: Does nothing except logging.

`Tkinter::Label::~Label` () []

Destructor: Does nothing except logging.

void `Tkinter::Label::init` **(Tk_Object&)** []

Creates the QLabel widget and sets the widget of the Tk_Object p as its parent.

void `Tkinter::Label::init` **(Tk_Object&, Keyword&)** []

Creates the QLabel widget and sets the widget of the Tk_Object p as its parent. This also reads the keyword arguments to the method.

void `Tkinter::Label::pack` () []

Packs the widget using the pack layout manager. This is poorly supported. Use the grid-methods instead.

Reimplemented from *Tk_Object*

E.8 class Tkinter::Radiobutton

A radiobutton that can be assigned to a function and grouped with other radiobuttons.

`#include <c_tkinter.h>`
Inherits: Tk_Object

public

()

Public Methods

Protected Methods

Protected Members

Detailed Description

A radiobutton that can be assigned to a function and grouped with other radiobuttons. QRadioButton is attached to this class.

Member Documentation

`Tkinter::Radiobutton::Radiobutton` () []

Constructor: Does nothing except logging.

`Tkinter::Radiobutton::~Radiobutton` () []

Destructor: Does nothing except logging.

void `Tkinter::Radiobutton::init` **(Tk_Object&)** []

Creates the QFrame widget and sets the widget of the Tk_Object p as its parent.

void `Tkinter::Radiobutton::init` **(Tk_Object&, Keyword&)** []

Creates the QFrame widget and sets the widget of the Tk_Object p as its parent. This also reads the keyword arguments to the method.

void `Tkinter::Radiobutton::pack` **()** []

Packs the widget using the pack layout manager. This is poorly supported. Use the grid-methods instead.

Reimplemented from *Tk_Object*

E.9 class Tkinter::StringVar

A string objects that can be assigned other widgets.

`#include <c_tkinter.h>`

Inherits: Tk_Object

public

()

Public Methods

Public Slots

Protected Methods

Detailed Description

A string objects that can be assigned other widgets. QString is attached to this class.

Member Documentation

`Tkinter::StringVar::StringVar` () []

Constructor: Does nothing except logging.

`Tkinter::StringVar::~StringVar` () []

Destructor: Does nothing except logging.

void `Tkinter::StringVar::init` () []

Does nothing except logging.

void `Tkinter::StringVar::pack` () []

Does nothing.
Reimplemented from *Tk_Object*

E.10 class Tkinter::TextObject

The super class of the widgets that can have a string variable assigned to them.
Abstract class.
`#include <c_tkinter.h>`
EntryLabel

Public Methods

Detailed Description

The super class of the widgets that can have a string variable assigned to them.

Member Documentation

void *Tkinter::TextObject::setText* **(string s)** [**pure virtual**]

Sets the text of the string to s.

string *Tkinter::TextObject::getText* () [**pure virtual**]

Returns the test of the string.

E.11 class Tkinter::Tk

The main class in the hierarchy of widgets.

`#include <c_tkinter.h>`

Inherits: Tk_Object

public

()

Public Methods

Protected Methods

Detailed Description

The main class in the hierarchy of widgets. QApplication is attached to this class.

Member Documentation

`Tkinter::Tk::Tk` () []

Constructor: Does nothing except logging.

`Tkinter::Tk::∼Tk` () []

Destructor: Does nothing except logging.

void `Tkinter::Tk::adjust` () []

Adjusts the widget's frame accordning to it's recommended size.

Reimplemented from *Tk_Object*

E.12 class Tkinter::Tk_Object

The super class of all widgets in this namespace.
Abstract class.
`#include <c_tkinter.h>`
Inherits: QObject

public

(unknown)
ButtonEntryFrameLabelRadiobuttonStringVarTk

Public Methods

Protected Methods

Detailed Description

The super class of all widgets in this namespace. This defines common methods and variables. It inherits QObject for Qt-support.

Member Documentation

QGridLayout* `Tkinter::Tk_Object::getGridLayout` **()** [**virtual**]

Returns the Gridlayout of the widget if one has been created. If no layout has been made, one is created by this method.

QWidget* *`Tkinter::Tk\_Object::getWidget`* **()** [**pure virtual**]

Returns the Qt widget of the current class.

void *`Tkinter::Tk\_Object::pack`* **()** [**pure virtual**]

Packs the widget using the pack layout manager. This is poorly supported. Use the grid-methods instead.

void `Tkinter::Tk_Object::adjust` () [**virtual**]

Adjusts the widget's frame accordning to it's recommended size.

void `Tkinter::Tk_Object::error` (**string s**) [**protected**]

Logging method: This prints the string s out to standard output if loglevel is set to 2 or lower.

void `Tkinter::Tk_Object::warn` (**string s**) [**protected**]

Logging method: This prints the string s out to standard output if loglevel is set to 3 or lower.

void `Tkinter::Tk_Object::info` (**string s**) [**protected**]

Logging method: This prints the string s out to standard output if loglevel is set to 4 or lower.

void `Tkinter::Tk_Object::debug` (**string s**) [**protected**]

Logging method: This prints the string s out to standard output if loglevel is set to 5 or lower.

void `Tkinter::Tk_Object::setLoglevel` (**int l**) [**protected**]

Sets the loglevel. 1-Critical, 2-Error, 3-Warn. 4-Info, 5-Debug

int `Tkinter::Tk_Object::getLoglevel` () [**protected**]

Returns the loglevel.

E.13 class Variable

Used as data type of variables of unknown type and in lists and dictionaries.

```
#include <c_variable.h>
```

Public Methods

Detailed Description

Used as type for variables when the actual type is unknown. This can store any type of variable.

Version:

1.0

Author:

Martin Jensen

Member Documentation

void `Variable::setValue` (void*, int) []

Sets the value to val. The t parameter specifies what kind of object it is.

void `Variable::setInt` (int) []

Sets the value of the Variable to val and sets the type to integer.

void `Variable::setString` (string) []

Sets the value of the Variable to val and sets the type to string.

void `Variable::setDouble` (double) []

Sets the value of the Variable to val and sets the type to double.

void `Variable::setPyList` (void*) []

Sets the value of the Variable to val and sets the type to PyList.

See also: PyList

void `Variable::setPyDict` **(void*)** []

Sets the value of the Variable to val and sets the type to PyDict.
See also: PyDict

void `Variable::setVariable` **(void*)** []

Sets the value of the Variable to val and sets the type to Variable.

int `Variable::getInt` **()** []

Returns an integer represenatation of the variable.

double `Variable::getDouble` **()** []

Returns a double represenatation of the variable.

void* `Variable::getObject` **()** []

Returns the Variable as a unknown object (void*).

string `Variable::getString` **()** []

Returns a string represenatation of the variable.

Variable `Variable::getVariable` **(int)** []

This method expects that the Variable is a List. It returns the content of the list in posistion i.

Variable `Variable::getVariable` **(string)** []

This method expects that the Variable is a Dictionary. It returns the content of the dictionary in place i.

int `Variable::getType` **()** []

Returns the type the variable currently have

string `Variable::toString` **()** []

Returns a string representation of the variable.

Variable `Variable::operator=` **(int)** []

This overloads the assignment operator so you can assign an integer to a Variable.

Variable `Variable::operator=` **(string)** []

This overloads the assignment operator so you can assign a string to a Variable.

Variable `Variable::operator=` **(double)** []

This overloads the assignment operator so you can assign a double to a Variable.

double `Variable::operator/` **(double)** []

This overloads the / operator so you can use the Variable in dividing operation.

E.14 class PyDict

Author: Martin Jensen Project: qGen Filename: c_list.

```
#include <c_dict.h>
```

Public Methods

Detailed Description

Author: Martin Jensen Project: qGen Filename: c_list.h Version: 1.0 Changed: 2004-11-29 05:00

This file contains the PyList-class used to emulate the behavior of Python-lists in C++

Member Documentation

void `PyDict::init` **()** []

void `PyDict::setValue` **(string, string)** []

Assigns the value val to the dictionary with the specified key. The value is of type string.

void `PyDict::setValue` **(string, int)** []

Assigns the value val to the dictionary with the specified key. The value is of type integer.

void `PyDict::setValue` **(string, double)** []

Assigns the value val to the dictionary with the specified key. The value is of type double.

void `PyDict::setValue` **(string, Variable&)** []

Assigns the value val to the dictionary with the specified key. The value is of type Variale.

See also: Variable

void `PyDict::setValue` **(string, PyDict&)** []

Assigns the value val to the dictionary with the specified key. The value is of type PyDict.

See also: PyDict

void `PyDict::setValue` **(string, PyList&)** []

Assigns the value val to the dictionary with the specified key. The value is of type PyList.

See also: PyList

int `PyDict::getInt` **(string)** []

Returns the element with the key 'key' as an integer.

string `PyDict::getString` **(string)** []

Returns the element with the key 'key' as a string.

double `PyDict::getDouble` **(string)** []

Returns the element with the key 'key' as a double.

char `PyDict::getChar` **(string)** []

Returns the element with the key 'key' as a character.

PyList `PyDict::getPyList` **(string)** []

Returns the element with the key 'key' as a PyList.

PyDict `PyDict::getPyDict` **(string)** []

Returns the element with the key 'key' as a PyList.

Variable `PyDict::getVariable` **(string)** []

Returns the element with the key 'key' as a Variable.

Variable `PyDict::getVariable` **(Variable&)** []

Returns the element with the string representation of the parameter 'var' as a Variable.

int `PyDict::size` **()** []

Returns the number of elements in the dictionary.

map<string, Variable>* `PyDict::getDict` **()** []

Returns the map of strings and Variables that represents the dictionary itself.

string `PyDict::toString` **()** []

Traverses through the dictionary and returns a textual representation of it. It could look something like this: \lbrace'key' : 'door', 'list' : [1,3,"tre"]\rbrace

E.15 class PyList

Used to emulate Python lists.

```
#include <c_list.h>
```

Public Methods

Detailed Description

Used to emulate Python lists. The list consist of a vector of Variables. The Variables can be of any type @see Variable.

Version:

1.0

Author:

Martin Jensen

Member Documentation

void `PyList::init` **()** []

void `PyList::append` **(int)** []

Appends an integer to the list.

void `PyList::append` **(string)** []

Appends a string to the list.

void `PyList::append` **(double)** []

Appends a double to the list.

void `PyList::append` **(PyList&)** []

Appends a PyList to the list.

void `PyList::append` **(PyDict&)** []

Appends a PyDict to the list. This class represents a Python dictionary.
See also: PyDict

void `PyList::append` **(Variable&)** []

Appends a Variable to the list. This is a general datatype.
See also: Variable

void `PyList::setValue` **(int, int)** []

Sets the element in place i to the integer value val.

void `PyList::setValue` **(int, string)** []

Sets the element in place i to the string value val.

void `PyList::setValue` **(int, double)** []

Sets the element in place i to the double value val.

int `PyList::getInt` **(int)** []

Returns the integer representation of the element in place i.

string `PyList::getString` **(int)** []

Returns the string representation of the element in place i.

double `PyList::getDouble` **(int)** []

Returns the double representation of the element in place i.

double `PyList::getDouble` **(Variable)** []

Returns the double representation of the element in place var. The parameter here is a variable. The integer representation of this Variable is used to make the lookup.
See also: Variable

char `PyList::getChar` **(int)** []

Returns the char representation of the element in place i.

PyList `PyList::getList` **(int)** []

Returns the PyList representation of the element in place i.

string `PyList::toString` **()** []

Returns a texual representation of the entire list. Like this [1,3,4]

PyList `PyList::operator+` **(PyList&)** []

This overloads the + operator so you can concatenate two list by adding them together.

PyList `PyList::operator*` **(int n)** []

This overloads the * operator. This creates a list with all elements of the initial list repeated n times.

Variable `PyList::getVariable` **(int)** []

Returns the variable at position i in the list.

Variable `PyList::initialize` **()** []

Initializes the iterator and returns the first element of list.

bool `PyList::hasMore` **()** []

Returns true if the list contains more elements, false is not.

Variable `PyList::nextElement` **()** []

Returns the next element of the list and updates the pointer to point at the next element.

E.16 class PySys

Adds support for command line arguments to the programs.

```
#include <c_sys.h>
```

Public Methods

Public Members

Detailed Description

Adds support for command line arguments to the programs.

Member Documentation

`PySys::PySys` **() []**

Empty constructor. Does nothing.

`PySys::PySys` **(int argc, char* arg[]) []**

Constructor that reads in the arguments from the argumentlist that is sent to the program.

PyList `PySys::argv`

E.17 class Keyword

Used when keywords are sent as argument to a function in Python.

```
#include <c_keyword.h>
```

Public Methods

Detailed Description

Used when keywords are sent as argument to a function in Python. An instance of this class is populated with the keywords and sent as the argument to the corresponding C++ function.

Member Documentation

void `Keyword::add` **(string, void*) []**

Adds a keyword argument to the keychain with the key key and value value.

void `Keyword::add` **(string, int) []**

Adds a keyword argument to the keychain with the key key and value value.

void `Keyword::add` **(string, string) []**

Adds a keyword argument to the keychain with the key key and value value.

void `Keyword::add` **(string, void value()) []**

Adds a keyword argument to the keychain with the key key and value value.

void* `Keyword::get` **(string)** []

Returns the element of the keychain with key key.

map<string, void*> `Keyword::getKeychain` **()** []

Returns the entire keychain with all keywords.

E.18 namespace builtin

Contains C++ implementation of some of the most commonly used built in functions in Python.

Public Methods

Detailed Description

Contains C++ implementation of some of the most commonly used built in functions in Python.

Version:

1.0

Author:

Martin Jensen

Member Documentation

int `builtin::c_int` **(Variable var)** []

Returns an integer representation of the Variable given as argument. This is the C++ implementation of the Python int()-function.

int `builtin::c_int` **(string s)** []

Returns an integer representation of the string given as argument. It uses the atoi function to convert the string. This is the C++ implementation of the Python int()-function.

int `builtin::c_int` **(int i)** []

Returns the integer given as argument. This is the C++ implementation of the Python int()-function.

int `builtin::c_int` **(double d)** []

Returns an integer representation of the double given as argument. This is the C++ implementation of the Python int()-function.

string `builtin::str` **(char* s)** []

Returns a string representation of the char* given as argument. This is the C++ implementation of the Python str()-function.

string `builtin::str` **(string s)** []

Returns a the string gives as argument. This is the C++ implementation of the Python str()-function.

string `builtin::str` **(double d)** []

Returns a string representation of the double given as argument. The method uses the a ostingstream to convert the double to string. This is the C++ implementation of the Python str()-function.

string `builtin::str` **(int i)** []

Returns a string representation of the integer given as argument. The method uses the a ostingstream to convert the integer to string. This is the C++ implementation of the Python str()-function.

string `builtin::str` **(Variable var)** []

Returns a string representation of the Variable given as argument. The method calls upon the Variables toString() method to convert it. This is the C++ implementation of the Python str()-function.

double `builtin::c_float` **(string s)** []

Returns a double representation of the string given as argument. The method uses the atof()-method to convert it. This is the C++ implementation of the Python float()-function.

double `builtin::c_float` **(double d)** []

Returns the double given as argument.

double `builtin::c_float` **(Variable var)** []

Returns a double representation of the Variable given as argument. The method calls upon the Variables getDouble()-method to convert it. This is the C++ implementation of the Python float()-function.

PyList `builtin::range` **(int from, int to, int inc = 1)** []

Creates a PyList of integers. This emulates the behaviour of the Python range()-method. The arguments are the starting integer, ending integer and the increment.

PyList `builtin::range` **(int to)** []

This is the overloaded function of the above function. It assumes the starting integer to be 0 and the increment to be 1.

E.19 namespace math

This namespace interfaces some functions of the <math.

Public Methods

Detailed Description

This namespace interfaces some functions of the <math.h> library of C++. The namespace has as function to map the Python math-functions to the ones in the C++ library.

Member Documentation

double `math::acos` **(double d)** []

The function returns the angle whose cosine is x, in the range [0, pi] radians.

double `math::asin` **(double d)** []

The function returns the angle whose sine is x, in the range [-pi/2, +pi/2] radians.

double `math::atan` **(double d)** []

The function returns the angle whose tangent is x, in the range [-pi/2, +pi/2] radians.

double `math::atan2` **(double d, double dd)** []

The function returns the angle whose tangent is y/x, in the full angular range [-pi, +pi] radians.

double `math::ceil` **(double d)** []

The function returns the smallest integer value not less than x.

double `math::cos` **(double d)** []

The function returns the cosine of x for x in radians. If x is large the value returned might not be meaningful, but the function reports no error.

double `math::exp` **(double d)** []

The function returns the exponential of x, e^x.

double `math::fabs` **(double d)** []

The function returns the absolute value of x, —x—.

double `math::floor` **(double d)** []

The function returns the largest integer value not greater than x.

double `math::fmod` **(double d, double dd)** []

The function returns the remainder of x/y, which is defined as follows:
- If y is zero, the function either reports a domain error or simply returns zero. - Otherwise, if 0 <= x, the value is x - i*y for some integer i such that: 0 <= i*—y— <= x < (i + 1)*—y— - Otherwise, x < 0 and the value is x - i*y for some integer i such that: i*—y— <= x < (i + 1)*—y— <= 0

double `math::log` **(double d)** []

The function returns the natural logarithm of x.

double `math::log10` **(double d)** []

The function returns the base-10 logarithm of x.

double `math::pow` **(double d, double dd)** []

The function returns x raised to the power y, x^y.

double `math::sin` **(double d)** []

The function returns the sine of x for x in radians. If x is large the value returned might not be meaningful, but the function reports no error.

double `math::sinh` **(double d)** []

The function returns the hyperbolic sine of x.

double `math::sqrt` **(double d)** []

The function returns the square root of x, $x^(1/2)$.

double `math::tan` **(double d)** []

The function returns the tangent of x for x in radians.If x is large the value returned might not be meaningful, but the function reports no error.

double `math::tanh` **(double d)** []

The function returns the hyperbolic tangent of x.

E.20 namespace py2c

This is meant to contain functions that are needed by qGen to make the target code compile.

Public Methods

Detailed Description

This is meant to contain functions that are needed by qGen to make the target code compile. These are not function that the source code calls. So far this only consists of methods that converts elements to PyLists.

Member Documentation

PyList `py2c::toPyList` **(string s)** []

Takes a string as parameter and creates a PyList with the characters from the string as the elements. This is used when looping over a string in a for-loop.